CERIAS Tech Report 2009-34
Architectural approaches for code injection defense at the user and kernel levels
by Riley, Ryan
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ryan Denver Riley

Entitled Architectural Approaches for Code Injection Defense at the User and Kernel Levels

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Dongyan Xu
Chair

Dr. Eugene Spafford

Dr. Xuxian Jiang

Dr. Cristina Nita-Rotaru

Dr. Sonia Fahmy

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. Dongyan Xu

Dr. Xuxian Jiang

Approved by: Dr. Aditya Mathur July 22, 2009
Head of the Graduate Program Date

Graduate School Form 20
(Revised 6/09)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
Architectural Approaches for Code Injection Defense at the User and Kernel Levels

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Ryan Denver Riley
Printed Name and Signature of Candidate

07/22/2009
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

ARCHITECTURAL APPROACHES FOR CODE INJECTION DEFENSE AT THE USER AND KERNEL LEVELS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ryan D. Riley

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2009

Purdue University

West Lafayette, Indiana

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

To the one who died that I might live.

Soli Deo Gloria

ACKNOWLEDGMENTS

I would like to start by thanking my advisors, Professors Dongyan Xu and Xuxian Jiang. They have invested countless hours into me, both as a researcher and as a human being. Their support and care has been invaluable, and they have continually looked out for my best interests, even when I was not. Without them I would have never made it through, and I certainly wouldn't have published anything. Professor Eugene Spafford has consistently reminded me to look at the science of what I'm doing, and to look to the past for inspiration. Professor Cristina Nita-Rotaru taught countless classes I was in and provided a service very few people do – she told me what she really thought. Professor Sonia Fahmy has faithfully served on my committee and given me invaluable advice during my job search. Professor Doug Comer and Professor David Yau have each made significant impacts on me by causing me to think deeply about quality teaching as well as my own teaching philosophy.

My lab mates, Junghwan Rhee, Ardalan Kangarlou, Sahan Gamage, and Zhiqiang Lin have provided countless hours of brainstorming, encouragement, and even camaraderie. I will sorely miss our random conversations about topics ranging from research to real life. Both Xuxian Jiang and Paul Ruth, graduates from our lab, provided me important insights into what it takes to actually graduate and find a job. Jeff Turkstra has been with me since the beginning of my Purdue career and the friendship we formed will last a lifetime – I'm happy to report that I beat him to our third and final graduation.

The staff in the Computer Science department has been invaluable in ensuring I made it through unscathed. Mike Motuliak has provided better hardware support than I ever imagined possible. He even replaced a capacitor on a broken motherboard when I had a deadline looming. Brian Board has spent far more time than I deserve fixing my various networking problems and getting new computers online fast. Linda

Byfield ensured I got reimbursed for all of my travel – even when I made that difficult. Amy Ingram has shown excessive patience in answering my questions while providing an encouraging smile.

My gorgeous wife Betsy may have come to me late in my graduate school career, but no person has brought me more encouragement and joy in this journey. There is no one else I would rather have by my side in the trials and troubles of life. She is far more precious to me than she will ever know.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Riley, Ryan D. Ph.D., Purdue University, August 2009. Architectural Approaches for Code Injection Defense at the User and Kernel Levels. Major Professors: Dongyan Xu and Xuxian Jiang.

Code injection attacks, despite being well researched, continue to be a problem today. Modern architectural solutions such as the execute-disable bit have been useful in limiting the attacks, however they enforce program layout restrictions and can often still be circumvented by a determined attacker. In this dissertation, we analyze the code injection problem from the perspective of a vulnerable system's memory architecture. We propose an alternative memory architecture, the split memory architecture (SMA), which is not susceptible to code injection attacks. This memory architecture can be implemented either in software running on a von Neumann memory architecture or through slight modifications to the von Neumann architecture. The SMA is also able to support the execution of unmodified programs and operating systems designed and compiled for a von Neumann system.

We demonstrate the efficacy of the SMA approach at the user-level by presenting the design, implementation, and evaluation of an operating system level patch to run a process inside an SMA. The results show that the system is able to prevent a variety of code injection attacks while imposing less than 20% overhead on average.

We also demonstrate an SMA at the kernel-level with NICKLE, an instantiation of an SMA in a virtual machine monitor (VMM). We use NICKLE to verify the applicability of the SMA design to the prevention of code injection based kernel rootkits. Our evaluation reveals that NICKLE is able to prevent the execution of these rootkits while imposing less than 10% overhead to QEMU. The VMM-based

SMA is also used as the basis for a rootkit profiler named PoKeR, which is able to help human experts determine the behavior of a rootkit.

Our results reveal that the SMA can be a solution for preventing code injection attacks in both user-level applications and the operating system kernel.

1 INTRODUCTION

1.1 Background and Problem Statement

Code injection attacks, in their various forms, have been in existence and been an area of consistent research for a number of years [1, 2]. A code injection attack is a method whereby an attacker inserts malicious code into a running computing system and transfers execution to his malicious code. In this way he can gain control of a running process or operating system because his injected code will run at the same privilege level as the entity being attacked.

A widely utilized memory architecture in desktop computers is known as a von Neumann memory architecture [3]. In this architecture code and data share a common memory space. As such, data injected by an attacker can be executed as code by the processor. A different type of architecture, the Harvard [4, 5] architecture, enforces a strict separation of code and data, making it infeasible for an attacker to inject data into the system and force execution of it. As such, a Harvard architecture does not have the "features" required for a code injection attack to succeed.

Architectural code injection prevention techniques, such as the execute-disable bit and segmentation, rely on a separation of code and data, either into pages or segments. In effect, these techniques force a Harvard like separation of information while executing on a von Neumann architecture. This forced separation of code and data may prevent these techniques from being properly used in the context of some operating system kernels.

In this dissertation we present the split memory architecture (SMA), a memory architecture that has the code injection immunity benefits of a Harvard architecture and yet can be constructed for software running on a von Neumann system. As such, the thesis for this work is as follows: *The split memory architecture can prevent*

code injection attacks that a standard von Neumann architecture cannot, and it can do so while incurring reasonable overhead. In addition, the split memory architecture can be constructed in software on a von Neumann system and existing, unmodified programs and operating systems can be executed on it.

1.2 Contributions

The contributions of this dissertation are as follows:

- **Analysis of Code Injection and Memory Architectures** An analysis of the code injection problem with respect to the system's underlying memory architecture is conducted. The properties of the memory architectures related to permitting or denying code injection is enumerated and discussed. A new memory architecture, named a split memory architecture, not susceptible to code injection attacks, is proposed.

- **User Level Code Injection Prevention** An operating system level approach to implementing an SMA on an unmodified Intel x86 processor is given. Using this architectural approach, code and data become physically together but function as if they are separate, and injected code is unable to be fetched for execution. The system is also able to protect pages that contain both code and data, something which existing page-level execution prevention techniques are unable to do. In addition, various techniques for responding to code injection attacks are also presented.

- **Kernel Level Code Injection Prevention** While a significant amount of research has been done regarding code injection prevention at the user-level, relatively little work has been done at the kernel level. As such, the main contributions of this dissertation are in the area of kernel level code injection prevention. An analysis of kernel level code injection attacks and the assumptions they make regarding the underlying memory architecture is presented.

Specifically, the attack case to be considered will be kernel level rootkits employing code injection to accomplish their means. Building on the concepts of generalized code injection, a theoretical understanding of kernel level code injection and prevention is presented. A method for constructing an SMA using a virtual machine monitor is described. This method is then instantiated in multiple virtual machine monitors that are used to verify the effectiveness of an SMA against kernel rootkit attacks. Our system is among the first capable of preventing kernel-level code injection attacks, and has sparked additional research into more effective rootkit attacks [6].

- **Kernel Rootkit Profiling** The SMA is not only applicable to code injection prevention, it also provides unique benefits that allow the profiling of injected code. Accordingly, an SMA based system for analyzing the behavior of injected kernel rootkit code will be described. At the core of this system, an SMA provides a convenient mechanism for identifying injected rootkit code before it is executed. This "right-before" timing allows us to begin profiling as soon as the kernel portion of the attack begins. Techniques and algorithms for monitoring and profiling this malicious code for the purpose of quickly ascertaining what a rootkit does will also be presented. We propose the combat tracking technique to determine the identity of targeted kernel objects, even when they are dynamically located. This system is one of the first in the growing area of kernel rootkit profiling.

1.3 Terminology

This section defines terminology used throughout this dissertation.

- **Attacker** Our definition of an attacker is adapted from the definition of penetration found in [7]. An attacker is a person, organization, or computer program that is attempting to obtain unauthorized access to files, programs, or the control state of a computer system. We further assume that the attacker

is attempting to gain this access remotely. We do not consider physical access, social engineering, or similar attacks.

- **Control-flow Hijacking** Control-flow hijacking is an attack strategy wherein an attacker modifies the control-flow of a running computing system by causing direct modifications to the system's program counter register. A buffer overflow attack, for example, is a type of control-flow hijacking.

- **Code Injection** Code injection is the process of an attacker adding new code (ranging from machine instructions to a high-level scripting language) to a computer system. Unless otherwise noted, in this dissertation we will only use code injection to refer to the injection of machine instructions.

- **Code Injection Attack** A code injection attack occurs when an attacker combines code injection and control-flow hijacking to cause new, injected code to be executed on a computing system.

- **Rootkit** We adapt our definition of rootkit from Greg Hoglund et al. [8]. A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer. This presence is not authorized by the computer's administrator. An attacker uses a rootkit on an operating system to maintain access as long as possible without being detected. Rootkits will frequently hide files, processes, network activity, and log entries. Rootkits exist for a variety of operating systems. In this dissertation we are concerned with **kernel rootkits**, which are rootkits that accomplish their goals by modifying the running operating system kernel.

- **Virtualization** When we refer to virtualization we are referring to operating system virtualization. Our practical use of virtualization in this dissertation is limited to x86 systems, although the theoretical design is applicable to multiple underlying architectures. We form our definition of virtualization and virtual machines by combining definitions from Meyer et al. [9] as well as Goldberg [10]:

The expression, "virtual machine," is now generally accepted as a software replica of a complete computer system. It consists of a data structure describing the memory size and the input/output configuration of the simulated system. In these systems, much of the software for the simulated machine executes directly on the hardware. Systems of this kind are called virtual machine systems, the simulated machines are called **virtual machines** (VMs), and the simulator software is called the **virtual machine monitor** (VMM).

- **Authorized Code / Authenticated Code** When we refer to authorized or authenticated code we are referring to a body of program or operating system machine code that has been verified to be unmodified and un-appended to when compared to some prior state. For example, an operating system's running code may be authenticated by verifying that it matches, byte-for-byte, the code that was initially produced by the compiler. Given that very large code bases (such as OS kernels) are not usually formally verified, we do not consider the formal correctness of the code or provide a guarantee that it does not contain bugs or security vulnerabilities. Instead, we rely on a developer or administrator to designate a piece of code as authorized.

1.4 Execution Model

In this work we assume a uni-processor system using a fetch-execute cycle to execute instructions stored in a physical memory space which employs virtual memory using page tables. The system supports multiple processes, each of which has its own virtual memory space defined by a page table. Figure 1.1 illustrates this execution model. The model is based on models described in [11] and [12]. A basic overview of this model is described here, for details of any of these concepts see the previous two references.

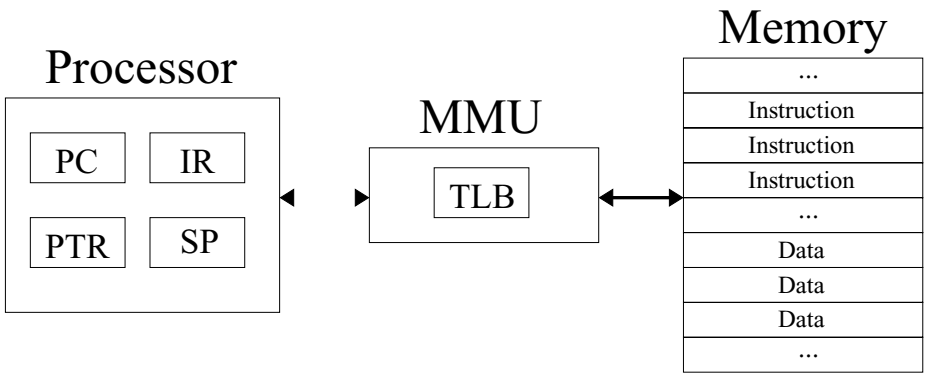


Figure 1.1. Execution model

1.4.1 Processor

The processor has four control registers: a program counter (PC), the instruction register (IR), the page table register (PTR), and the stack pointer (SP). The PC contains the address of the next instruction to be fetched for execution. The IR contains the last instruction fetched for execution. The PTR contains the address of the current page table in memory. The SP points to the last item added to the stack. (Alternatively, it could point to the next available empty slot on the stack.) The stack itself grows from higher addresses to lower addresses (downward growth).

The processor fetches instructions from the system's memory for execution. This occurs during two cycles, a fetch cycle and an execute cycle. During the fetch cycle the instruction located at the address indicated by the PC is loaded into the instruction register and the PC is incremented to the address of the next instruction. During the execution cycle the instruction in the IR is decoded and executed. These two cycles may occur simultaneously, with one instruction being fetched and the other executed at the same time. A more complicated pipelining architecture may also be used [13]. During a function call, the current PC is pushed onto the stack before being modified to reflect the address of the function being called.

1.4.2 Memory

The processor interfaces to memory through the memory management unit (MMU) which translates virtual addresses into physical addresses. Pages are 4 kilobytes in size. The MMU makes use of the page table found at the address specified by the PTR. (Ensuring the correctness of the page table and the PTR is the responsibility of the operating system.) This procedure requires multiple accesses to memory to translate one virtual address. A small hardware cache called the translation lookaside buffer (TLB) is used to store recently accessed page table entries. Many page table lookups never need to go all the way to the page table, but instead can be served by the TLB. Each TLB entry corresponds to one entire memory page.

1.5 Dissertation Organization

This dissertation contains seven chapters. Chapter 2 contains an overview of code injection attacks as well as related work. Chapter 3 presents an analysis of the code injection problem with respect to a system's underlying memory architecture and proposes using a split memory architecture for code injection prevention on existing von Neumann systems. Chapters 4 and 5 demonstrate the effectiveness and performance of this architecture for protecting both the user and kernel levels. Chapter 6 presents a method for using this architecture to determine the behavior of injected rootkit code. In chapter 7 we summarize, draw some conclusions, and present future work.

2 ATTACK OVERVIEW AND RELATED WORK

In this chapter we give an overview of various code injection attacks and the defense mechanisms related to them.

2.1 Code Injection Attacks

In this section we will discuss various types and components of code injection attacks.

2.1.1 Control-flow Hijacking

In this work we define control-flow hijacking as an attack strategy wherein an attacker modifies the control-flow of a running computing system by causing direct modifications to the PC. We will discuss a number of control-flow hijacking techniques here.

A stack based buffer overflow occurs when a write to an array located on the stack goes beyond the allocated bounds of that array. Under this circumstance various control structures also stored on the stack, such as the return address, may be overwritten. When the function later returns, the modified return address will be loaded into the PC, causing modification to a program's control-flow. A description of stack based buffer overflows as well as exploitation techniques is available in [14]. The Morris Worm [1] made use of this type of attack. A variation of this attack modifies function pointers on the stack as a means to modify control flow.

The term heap overflow describes a vulnerability where a buffer allocated on the heap has data written beyond its bounds. Under this circumstance an attacker may modify function pointers in neighboring heap allocations or even modify heap control

data to cause the memory allocator to overwrite arbitrary memory addresses. A similar attack would be a double-free, wherein a program mistakenly frees the same heap variable twice. An attacker may be able to leverage this situation to modify arbitrary memory locations.

A format string vulnerability occurs when a buffer that is filled with data from an untrusted source (such as the network, a file, a user, etc.) is passed directly to a function in the `printf` family as the format string. An attacker can make use of this vulnerability to cause a program to display the values of arbitrary stack entries (data leakage) and even directly modify arbitrary memory addresses.

2.1.2 Payload Execution

Thus far we have discussed various ways an attacker may hijack the control-flow of a running system. We have not, however, discussed what an attacker may wish to execute (the payload) using this change of control flow.

The first, and most obvious, target for a hijacked instruction pointer would be new code that the attacker herself has written and added to the system's memory. (We will refer to this henceforth as *injecting* code.) Injecting code is not normally a difficult procedure, as a system's normal input routines (file I/O, network I/O, user I/O, etc.) can be used to cause arbitrary attack code to be written to memory. In the case of a stack based buffer overflow, for example, an attacker could add her code into the buffer itself, or if the buffer is too small it could be written to a heap based buffer using some other program functionality. Once an attacker has injected her code into the memory space, she can use any of the above control-flow hijacking techniques to cause it to be executed.

A different option for an attacker would be to point the instruction pointer to existing code. This attack, traditionally called a return-to-libc attack [15], makes use of a control-flow hijacking technique combined with a carefully crafted stack to use portions of existing code to accomplish an attacker's goals. In Shacham et al. [16]

this concept of "return oriented programming" was been shown to provide Turing completeness given a sufficiently large code base, specifically a large C library. This attack was further refined and generalized to other architectures as well [17].

2.1.3 Kernel Rootkits

A kernel rootkit is a program that modifies a running operating system kernel to maintain an unauthorized, undetectable presence on a computer system. Many kernel rootkits use code injection to modify the running kernel.

The control-flow hijacking and code injection techniques of these rootkits may differ slightly from those already mentioned. While a kernel rootkit may make use of traditional attack vectors such as a stack based buffer overflow, the rootkits discussed in this dissertation gain full read and write access to kernel memory using existing OS mechanisms.

Some rootkits are installed into the running kernel directly as loadable kernel modules. Using this technique, a rootkit is written and compiled as a runtime extension to the OS that is then loaded into the kernel's memory space and executed. Once running, the module has full access to all of kernel memory and can modify operating system data structures and function pointers directly. The rootkit may, for example, hijack control flow by modifying function pointer entries in the system call table.

Other rootkits are installed into the kernel from user-space by making use of a raw memory access device[1] such as `/dev/kmem`. Raw memory access devices exist to allow certain privileged programs to read and write kernel memory directly. A rootkit is able to use these devices to directly write new code into the kernel's memory space and modify function pointers to hijack control flow.

[1]This is not an actual device, it is implemented entirely in software. It is referred to as a device only because it uses the OS device driver interface.

2.1.4 Other Code Injection Attacks

If our definition of code injection were to be expanded, there are a number of other types of attacks which might also be considered code injection attacks. We mention these for completeness, but they are outside the scope of this dissertation.

- A non-control-data attack [18] is one that indirectly changes a program's control flow by modifying crucial control variables, such as variables used in the evaluation of conditionals.
- SQL Injection attacks [19] can occur when user input to a database-driven web application is not properly validated. Certain inputs can be used to pass arbitrary queries to the underlying database.
- Cross-site scripting (XSS) attacks [20] are another example of improper input validation. In this case, however, a malicious URL is used by an attacker to inject malicious javascript onto a legitimate website being viewed by a victim.

2.2 Deterrence Techniques

In response to the attack techniques discussed above, defense measures have been developed. We classify these into two major categories: compiler based approaches that focus on preventing control flow hijacking, and exection prevention techniques that focus on preventing the execution of attack code.

2.2.1 Compiler Approaches

There are a number of code injection prevention approaches that focus on modifying the compiler to produce a binary that is not susceptible to one or more control-flow hijacking techniques.

StackGuard [21] attempts to prevent the stack based buffer overflow by modifying the compiler to insert a randomly generated "canary value" on to the stack between

the return address and the locally allocated variables. This canary is then verified prior to returning from a function. If an attacker overflows a buffer to overwrite the return address, she will also need to overwrite the canary value. This technique makes a number of assumptions. First, it assumes that the attacker does not know the canary value. This may not be true, as other vulnerabilities such as a format string vulnerability (discussed below) can be used to reveal arbitrary memory locations. Another assumption is that the attacker must overflow a buffer to get to the return address, which may also be untrue. For example, an attacker could instead overflow the buffer into other pointers on the stack to cause valid code to later overwrite the return address directly.

Another compiler-based solution, Pro Police [22], builds on StackGuard's scheme by also rearranging local, stack-based variables to ensure that buffers are located at higher addresses than function pointers. The technique has been effective in practice, however it is unable to protect against format string vulnerabilities or heap based vulnerabilities. Another compiler based solution, PointGuard [23], extends the StackGuard concept to protect function pointers as well. PointGuard suffers from the same weaknesses as Pro Police. Yet another compiler based technique, Stack Shield [24] uses a separate stack for return addresses as well as adding verification of `ret` and `call` targets.

A number of compiler based approaches rely on bounds checking to ensure that data is not written beyond the end of a buffer. Jones et al. [25] produced a compiler that ensures the results of pointer arithmetic refer to the same object as the original pointer. Cash [26] is a compiler based approach that leverages segmentation at runtime to cause memory access faults when a buffer overflow occurs.

FormatGuard [27] attempts to mitigate format string vulnerabilities by verifying that the proper number of arguments is passed for a given format string. This technique will prevent a large number of attacks that involve overwriting arbitrary memory, but is not able to prevent all format string attacks.

A problem with these compiler based solutions is that they tend to only work against known hijacking techniques. That means that while they are effective in some cases, they may miss many of the more complicated attacks. Wilander et al. [28], for example, demonstrates that some of these techniques miss a fairly large percentage (45% in the best case) of attacks that were implemented as part of a buffer overflow benchmark.

2.2.2 Execution Prevention

Regardless of the control-flow hijacking technique, there are a number of works related to preventing an attacker from executing attack code even if control-flow hijacking takes place.

One technique makes use of non-executable memory pages. This protection can come in the form of hardware support or a software only patch. Hardware support has been put forth by both Intel and AMD that extends the page-level protections of the virtual memory subsystem to allow for non-executable pages. (Intel refers to this as the "execute-disable" (XD) bit [29].) It is commonly applied using the W⊕X principle: Program information is separated into code pages and data pages. The data pages (stack, heap, bss, etc) are all marked non-executable. At the same time, code pages are all marked read-only. In the event an attacker exploits a vulnerability to inject code, it is guaranteed to be injected on a page that is non-executable and therefore the injected code is never run. Microsoft makes use of this protection mechanism in Windows XP SP2 as a part of Data Execution Protection (DEP) [30]. This method is effective for traditional code injection attacks, however it requires hardware support to be of use. Legacy x86 hardware does not support this feature. This technique is also available as a software-only patch to the operating system that allows it to simulate the execute-disable bit through careful mediation of certain memory accesses. PaX PAGEEXEC [31] is an open source implementation of this technique that is applied to the Linux kernel. It functions identically to the hardware

supported version, however it also supports legacy x86 hardware because of being a software only patch. A similar technique [32] makes use of the concept of segmentation to split a program in various segments that have the appropriate permissions to prevent the execution of injected code. These techniques are effective for many of the traditional attacks, however attackers still manage to circumvent them [33].

CuPIDS [34] makes use of a "shadow process" to monitor the execution of a running process. It can help mitigate and recover from stack based buffer overflows by monitoring program execution during "unsafe" system calls. Salamat et al. [35] offers a solution to reverse the direction of stack growth in a program and run it side-by-side with the original to detect when buffer overflows occur.

Other prevention techniques use randomization to thwart an attacker's ability to find or write injected code. Address Space Layout Randomization (ASLR) [36–39] is a technique whereby various portions of a process's memory space are placed at random locations to lower the probability an attack will succeed. By randomizing the memory layout of a running process, ASLR makes it hard for attackers to accurately locate injected attack code or existing program code (e.g., *libc* functions), hence lowering the probability an attack will successfully hijack control flow. One disadvantage to this technique is that there is often not enough entropy in the randomization to prevent an attacker from "guessing" the correct address when a large number of attempts are allowed. Instruction Set Randomization (ISR) [40–42] is a method whereby the code space of a process is encrypted in memory with a secret key and then decrypted immediately before execution. In this scenario, an attacker needs to know the secret key to encrypt a malicious payload for injection.

In response to return-to-libc attacks, a technique known as Control Flow Integrity (CFI) [43] was developed. CFI rewrites assembly code to enforce that all control-flow changes fall within a determined control flow graph. The technique, while effective, can add significant overhead to a running entity. CuPIDS can also be used to help monitor control flow when the program being protected has been instrumented to pass function call information to the shadow monitoring process. ASLR is also available

as a defense technique here, as it makes it more difficult for an attacker to learn the address of existing library code.

2.3 Kernel Rootkit Defense

There is a significant amount of work related to kernel level rootkits. We will now discuss their detection, prevention, and profiling.

2.3.1 Rootkit Detection

Most work on kernel rootkits relates to detection. A rootkit detection system is one that analyzes an OS to look for symptoms that a rootkit has been installed into the kernel.

Petroni et al. [44] and Zhang et al. [45] propose the use of external hardware to retrieve a copy of the runtime OS memory image and detect possible rootkit presence by detecting certain kernel code integrity violations (e.g., rootkit-inflicted kernel code manipulation). Follow up work further identifies possible violations of semantic integrity of dynamic kernel data [46] or state based control-flow integrity of kernel code [47]. Generalized control-flow integrity [43] may have strong potential to be used as a prevention technique, but as yet has not been applied to kernel integrity. Other solutions such as Strider GhostBuster [48] and VMwatcher [49] target the self-hiding nature of rootkits and infer rootkit presence by detecting discrepancies between the views of the same system from different perspectives.

2.3.2 Rootkit Prevention

Providing a much stronger guarantee than rootkit detection, rootkit prevention has the goal of preventing the rootkit attack from happening.

Livewire [50], based on a software-based VMM, protects the guest OS kernel code and critical data structures from being modified. In many ways Livewire provides a

foundation upon which we build in Chapter 5. SecVisor [51] leverages new hardware extensions to enforce life-time kernel integrity and prevent the execution of unauthorized code using the W⊕X principle for the protection of an OS's memory space. SecVisor requires modification to OS kernel source code as well as recent hardware support for MMU and IOMMU virtualization.

The concept of verifying the integrity of code prior to execution has been used previously in techniques such as Microsoft's ActiveX Authenticode [52] and more recently driver signing [53]. The concept was originally proposed by Cohen [54, 55] in the form of an integrity shell. An integrity shell uses a cryptographic checksum to verify that a program about to be executed has not been modified, and offers various response modes if it has. The technique was proposed for virus prevention, but the principles are applicable at the kernel level as well.

Various forms of driver verification [56, 57] have also been proposed. These techniques are helpful in verifying the identity or integrity of the loaded code. However, a kernel-level vulnerability could potentially be exploited to bypass these techniques.

2.3.3 Rootkit Profiling

Rootkit profiling is the process of determining a rootkit's behavior. There are a few early works in this relatively new area.

Panorama [58] performs system-wide information flow tracking to understand how sensitive data (e.g., user keystrokes) are stolen or manipulated by malware. The underlying taint-based information flow techniques fundamentally suffer from control-flow evasion attacks [59] that directly break taint propagation. From another perspective, K-Tracer [60] combines backward and forward slicing techniques to understand kernel rootkit behavior. However, the slicing operation requires prior determination of the sensitive data on which to perform the slicing analysis. As a result, although it is capable of dealing with regular kernel rootkits that hijack system call table entries, it becomes less efficient to handle advanced ones such as DKOM-capable rootkits.

Several other approaches have recently been proposed to understand rootkit hooking behavior. HookFinder [61] analyzes a given rootkit sample and reports a list of kernel hooks that are being used by the rootkit. HookMap [62] instead systematically enumerates all of the kernel hooks that can be hijacked for rootkit-hiding purposes. These approaches mainly focus on one aspect of rootkit behavior, the hooking behavior. They miss, however, other aspects that are also important for rootkit profiling purposes.

2.4 Organization of Attacks and Defenses

Table 2.1 illustrates which attacks the various defense techniques discussed above are able to prevent. The attack techniques are duplicated to distinguish between attacks involving the execution of code injected by the attack ("New Code") and those involving a return-to-libc style attack ("Existing Code"). A checkmark is placed in a box when the defense technique in the left hand column can prevent the attack in the upper row. In cases where a defense technique can protect some instances of the specific attack but not all, a checkmark is still placed. (ASLR, for example, can be defeated by some advanced return address overflows, however it still receives a checkmark.) The chart illustrates a best case scenario for each defense scheme.

Table 2.1
Organization of attack and defense techniques

		New Code						Existing Code					
	User/ Kernel[a]	Ret Addr Overflow[b]	Pointer Overflow[c]	Format String	Heap Overflow	Double Free	Direct Mod[d]	Ret Addr Overflow	Pointer Overflow	Format String	Heap Overflow	Double Free	Direct Mod
StackGuard	U	✓						✓					
Pro Police	U	✓	✓					✓	✓				
Stack Shield	U	✓						✓					
PointGuard	U	✓	✓					✓	✓				
FormatGuard	U			✓						✓			
Jones et al. [25]	U	✓	✓		✓			✓	✓		✓		
XD Bit	U	✓	✓	✓	✓	✓	✓						
PAGEEXEC	U	✓	✓	✓	✓	✓	✓						
SEGMEXEC	U	✓	✓	✓	✓	✓	✓						
ISR	U	✓	✓	✓	✓	✓	✓						
ASLR	U	✓	✓		✓	✓		✓	✓		✓	✓	
CFI	U	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Salamat et al. [35]	U	✓	✓					✓	✓				
CuPIDS	U	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cash	U	✓	✓		✓			✓	✓		✓		
SecVisor	K	✓	✓	✓	✓	✓	✓						

[a]User/Kernel signifies whether the system was designed to prevent attacks at the user-level or kernel-level.
[b]Return Address Overflow refers to a stack based buffer overflow attack which modifies the return address.
[c]Pointer Overflow refers to instances where a buffer is overflowed and a program pointer is modified. This could occur on the stack or the heap.
[d]Direct Modification refers to attacks that modify memory directly, such as a kernel rootkit attack.

3 ARCHITECTURAL ANALYSIS OF CODE INJECTION ATTACKS

Code injection is a problem when a computing system permits code and data to share the same memory address space. Under such a system an attacker can inject his payload as data and later execute it as code. The underlying assumption relied on by attackers is that the processor's memory architecture does not strictly separate code and data or enforce a distinction between them.

For this reason, we approach the code injection problem by analyzing two different memory architectures and their susceptibility to code injection attacks. Next, we propose a new memory architecture that is not susceptible to code injection and discuss its benefits and features.

3.1 Basic Architectures

There are two models for memory architectures in existing computing devices that we will discuss with regards to code injection. The first, the von Neumann architecture, uses one memory space. The second, the Harvard architecture, makes use of two memory spaces. In this section we will describe each of these architectures as well as the susceptibility of their designs to code injection attacks.

3.1.1 The von Neumann Memory Architecture

The memory architecture code injection attacks implicitly rely on is known as a von Neumann memory architecture [3]. Under a von Neumann system there is one physical memory which is shared by both code and data. As a consequence of this, code can be read and written as data and data can be executed as code. Some systems will use segmentation or paging to help separate code and data from each

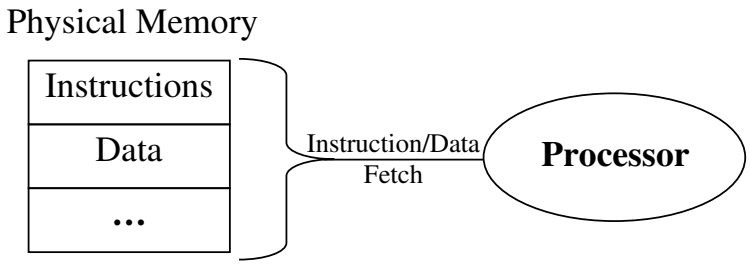


Figure 3.1. von Neumann architecture

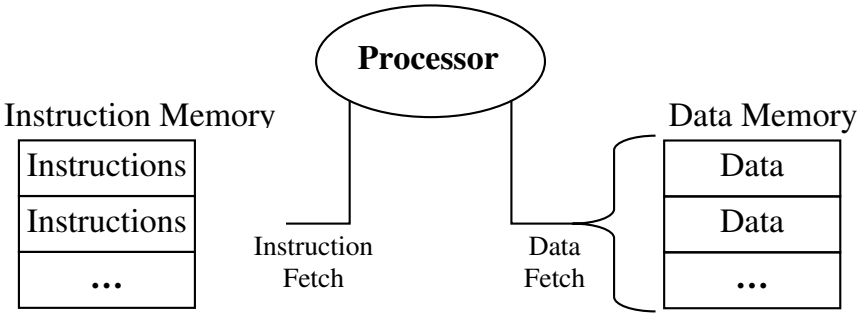


Figure 3.2. Harvard architecture

other or from other processes, but code and data end up sharing the same address space. Figure 3.1 illustrates a von Neumann architecture.

3.1.2 The Harvard Memory Architecture

An architecture found in some embedded processors [63] and operating systems [64] is known as a Harvard architecture [4, 5]. Under the Harvard architecture code and data each have their own physical address space. One can think of a Harvard architecture as being a machine with two different physical memories, one for code and another for data. Figure 3.2 shows a Harvard architecture.

The Harvard architecture's split memory model makes it immune to code injection attacks as defined in Chapter 2 because a strict separation between code and data is enforced at the hardware level. Any and all data, regardless of the source, is stored in a different physical memory from instructions. Instructions cannot be addressed as

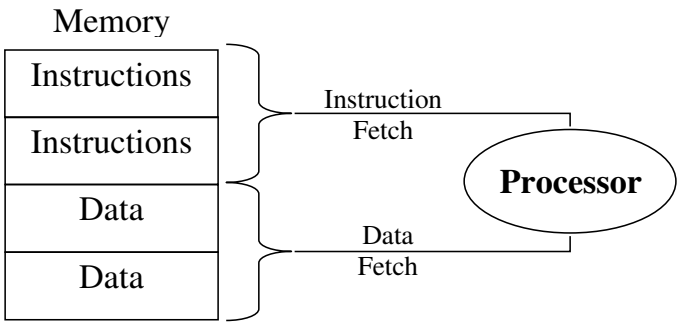


Figure 3.3. Split memory architecture

data, and data cannot be addressed as instructions. This means that the attacker is unable to inject any information whatsoever into the instruction memory and at the same time is unable to execute any code placed in the data memory. The architecture simply does not have the "features" required for a successful code injection attack.

3.2 Split Memory Architecture

When approaching the code injection problem from the memory architecture perspective, it would be desirable to have a memory architecture that has the code injection immunity benefits of the Harvard architecture and the versatility and install base of the von Neumann architecture. A new architecture is needed. In this work we propose a memory architecture known as the split memory architecture.

3.2.1 Overview

Figure 3.3 illustrates an SMA. This architecture bears resemblance to both of the previously discussed memory architectures. Like a von Neumann system, the architecture consists of only one physical memory space. Like a Harvard system, the processor is unable to fetch data as instructions or access instructions as data. Under this new architecture, instruction fetches are routed to one portion of physical memory while data accesses are routed to another. The SMA is designed to be implemented in

software running on a von Neumann memory architecture and support the execution of programs and operating systems designed for a von Neumann system.

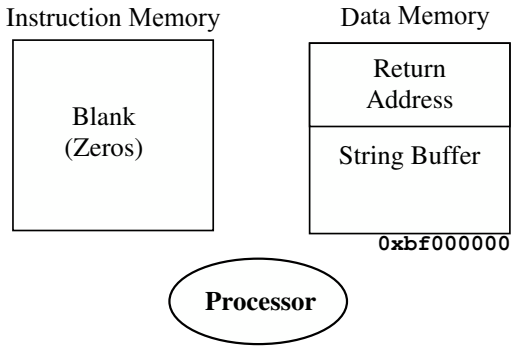
3.2.2 Effects on Code Injection

An SMA creates an environment wherein an attacker can exploit a vulnerable program and inject code into its memory space, but never be able to fetch it for execution. This is because the physical memory location that contains the data the attacker managed to write into the program is not accessible during an instruction fetch, as instruction fetches will be routed to an un-compromised memory location. To illustrate the effects of an SMA on code injection let us consider an example. A sample code injection attack attempt using a stack based buffer overflow on an SMA can be seen in Figure 3.4 and described as follows:

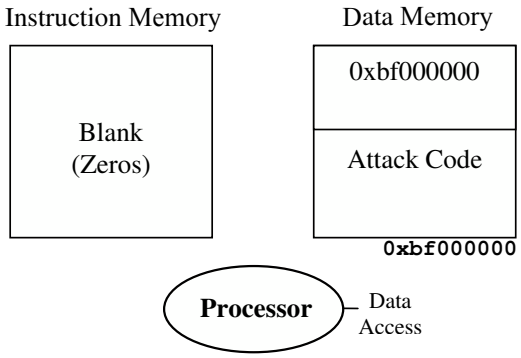
1. The attacker injects his code into a string buffer starting at address `0xbf000000`. The memory writes are routed to physical memory corresponding to data.
2. At the same time as the injection, the attacker overflows the buffer and changes the return address of the function to point to `0xbf000000`, the expected location of his malicious code.
3. The function returns and control is transferred to address `0xbf000000`. The processor's instruction fetch is routed to the physical memory corresponding to instructions.
4. The attacker's malicious code is not in the instruction memory (the code was injected as data and therefore routed to the data memory) and is not run. In all likelihood the instruction memory is empty (containing zeros) and the program simply crashes.

3.2.3 Further Applications

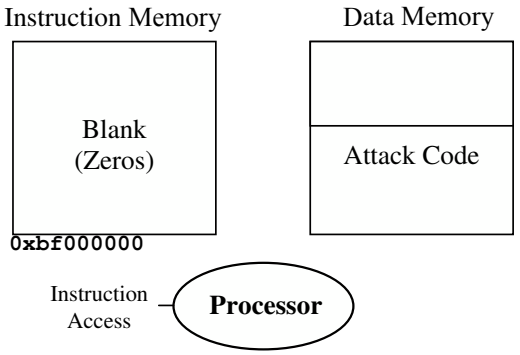
Using an SMA provides opportunities beyond that of prevention. Given that the architecture maintains separate copies of the code and data memories, it provides the



(a) Before the attacker injects code



(b) The injection to the data page



(c) The execution attempt that gets routed to the instruction page

Figure 3.4. Code injection attempt on the user-level virtual Harvard architecture.

unique opportunity for the comparison of the two memory spaces to preemptively detect the attack or even determine its behavior. Specific examples of such attack responses will be discussed in Chapters 4 and 6.

3.2.4 Limitations

When compared to the von Neumann architecture, the SMA has a gain of preventing the execution of data as code. There are a number of limitations as well:

- Self modifying code, including certain program language VMs such as Java, is not able to execute. The modifications would impact the data memory while execution would be attempted from the instruction memory.
- Under an SMA memory usage may be higher because two physical pages are required for each virtual page. In practice demand paging may be able to reduce this excess usage, but there will still be some additional memory usage.
- If being constructed and enforced using software, an additional performance penalty will be incurred. Details about this penalty are shown in Chapters 4 and 5.

3.2.5 Comparison to Other Architectural Approaches

There are a number of architectural security measures that can be used to prevent code injection attacks. Here we will discuss them and compare them with the SMA.

Segmentation, the concept of splitting a program into functional memory spaces, was first discussed by Holt [65] in 1961 as a method for splitting a program into parts to assist the loader in memory allocation. Segmentation gained popularity and became part of a number of computing systems such as Multics [66], the Burroughs B5000 [67], and the Intel 386 architecture [68]. Although it started as a mechanism for memory allocation, segmentation developed into a method for enforcing permissions as well. In a simple way, one could imagine using segmentation as a code injection

prevention technique by separating a program into segments for code and segments for data. Code segments would be read-only, and data segments would not be executable. To be protected by segmentation based protection, some programs would need to be heavily modified or recompiled to accommodate various segments. The SMA, as we will demonstrate, is capable of protecting unmodified programs that do not consider these sorts of permissions.

Paging, pioneered in the Atlas computer system [69], is the concept of splitting a program's flat memory space into smaller pieces and allocating them as needed. It can also be used as a code injection prevention mechanism. Both Intel and AMD have introduced the concept of non-executable pages to their architectures. As the name implies, a non-executable page is one which does not permit its content to be executed. By marking code pages read-only and data pages non-executable, code injection attacks can be mitigated, even for some unmodified programs that are not designed to make use of the protection. This technique is sometimes also referred to as W$\oplus$X (write exclusive-or execute) because no page should be both writable and executable at the same time.

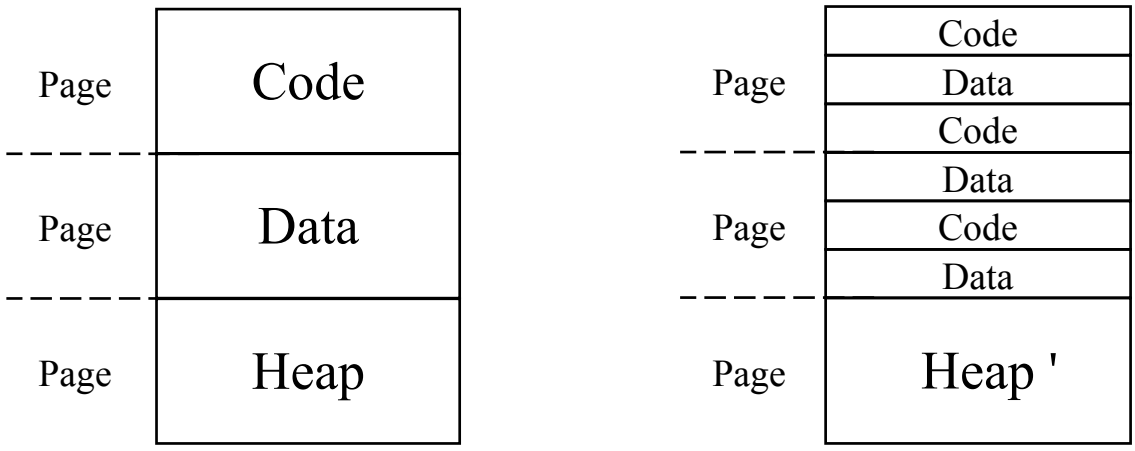
The use of non-executable pages has a few disadvantages when used for code injection prevention. First, it assumes that code and data are strictly separated into different pages. This assumption is not always correct. Chapter 5 will discuss instances of pages containing both code and data in an operating system kernel. Another disadvantage occurs because the permissions of memory pages often times must be dictated by the program itself. In such a scenario, an attacker may be able to change the permissions on existing pages or create a new memory allocation with the permissions she desires and inject code there. The SMA, in contrast, does have these same restrictions. As we will see in the user and kernel level experiments, mixed pages can be handled and the architecture (including permissions) is totally transparent to the protected entity, meaning that an attacker cannot modify the memory access rights.

4 SMA FOR USER-LEVEL CODE INJECTION DEFENSE

In this chapter, we will demonstrate the effectiveness and performance of the SMA when used to protect user-level applications under the Linux operating system running on an x86 processor [70]. While the high-level design is applicable to a variety of von Neumann processors, an Intel x86 processor is assumed.

4.1 Introduction

In this chapter we are concerned with addressing code injection attacks at the user level. The attacks described in Chapter 2 are all applicable at the user level. As we discussed in that chapter, an important defense technique is the hardware enabled execute-disable bit. While this technique is widely deployed and has proven to be effective, it has limitations. First, programs must adhere to the "code and data are



(a) Separate code and data pages

(b) Mixed code and data pages

Figure 4.1. Program memory layouts

always separated" model. See Figure 4.1(a) for an example of this memory layout. In the event a program has pages containing both code and data (see Figure 4.1(b)) the protection scheme cannot be used. Such "mixed pages" do exist in real-world software systems. For example, the Linux kernel uses mixed pages for both signal handling [71] as well as loadable kernel modules. A second problem with these schemes is that an advanced attacker can disable or bypass the protection bit using library code already in the process' address space and from there execute the injected code. Such an attack has been demonstrated for the Windows platform by injecting code into non-executable space and then using a well crafted stack containing a series of system calls and library functions to cause the system to create a new, executable memory space, copy the injected code into it, and then transfer control to it. One such example has been shown in [33].

It is these two limitations in existing page-level protection schemes (the forced code and data separation and the bypass methodology) that provide the motivation for our user-level work, which architecturally addresses the code injection problem at its core.

Our technique for SMA construction can be implemented as a lightweight, software-only patch for the operating system, and our implementation for the Intel x86 architecture incurs an average performance penalty between 10 and 20%. Such a software only technique is possible through careful exploitation of the two translation lookaside buffers (TLBs) on the x86 architecture to split memory in such a way that it enforces a strict separation of code and data memory. Furthermore, instead of letting the system crash when a code injection attack occurs, our technique is able to accommodate a number of *response modes* for attack monitoring and investigation. The experiments with a buffer overflow benchmark suite as well as five attacks on real-world software vulnerabilities successfully demonstrate the effectiveness of the proposed approach.

4.2 Challenges in Using an SMA on the x86

A goal of this chapter is to make use of an SMA on an Intel x86 processor. A technique for creating an SMA on the x86 is to make unconventional use of some x86 features to create the appearance of a memory that is split between code and data. Through careful use of the page table and the TLBs on x86, it is possible to construct an SMA at the process level using only operating system level modifications. No modifications need to be made to the underlying x86 architecture, and the system can be run on conventional x86 hardware without the need for hardware emulation.

In the following sections, we will further describe this technique for the x86 processor as well as its unique advantages. The realization of an SMA on other architectures (e.g., SPARC) will be discussed in Section 4.4.3.

4.3 Overview of the TLB on the x86

The are a few specifics regarding the TLB on the x86 that require more discussion than the overview provided in Section 1.4. The full details of paging on the x86 are available in the Intel manual [29].

On the x86 the loading of the TLB is managed automatically by the hardware, but removing entries from it can be handled by either software or hardware. The hardware, for example, will automatically flush the TLB when the OS changes the address of the currently mapped page table (such as during a context switch). Software can use the `invlpg` instruction to invalidate specific TLB entries when making modifications to individual page table entries to ensure that the TLB and page tables remain synchronized.

While the TLB is able to speed up virtual memory on the x86, one problem is that because it is limited in size, old entries are automatically removed when new ones come in. As a consequence of this, a program that makes many random data accesses may cause the TLB to flush entries related to code accesses, necessitating that they be reloaded if that code page is referenced again. To help prevent this problem, the

TLB is split into *two* TLBs in many x86 processor models, one for code and one for data. During normal operation one would want to ensure that the two TLBs do not contain conflicting entries (where one address could be mapped to different physical pages, depending on which TLB services the request).

4.4 Constructing an SMA

The key idea in our SMA construction technique is to exploit the dual TLB feature of the x86 architecture to route data accesses for a given virtual address to one physical page while routing instruction fetches to another. By desynchronizing the TLBs and having each contain a different mapping for the same virtual page, every virtual page may have two corresponding physical pages: One for code fetch and one for data access. In essence, a system is produced where any given virtual memory address could be routed to two possible physical memory locations. We will construct our SMA by splitting the individual pages in a process' memory space.

4.4.1 What to Split

Before we discuss the technical details behind successfully splitting a given page, it is important to note that different pages in a process' address space may be chosen to split based on how our system will be used.

One potential use of the system is to augment the existing non-executable page methods by expanding their protection to allow for protecting mixed code and data pages. Under this usage of the system, the majority of pages under a process' address space would be protected using the non-executable pages, while the mixed code and data pages would be protected using our technique. Note that this assumes we have a good understanding of the memory space of the program being protected. In addition, doing only partial protection using our technique may limit the use of the various response modes described in Section 4.4.5.

Another potential use of our system, and the one which we use in our prototype in Section 4.5, is to protect every page in a process' memory space. This is a more comprehensive type of protection than simply augmenting existing schemes. Note that in this case, more pages are chosen to be split and thus protected.

4.4.2 How to Split

Once it is determined which pages will be split, the technique for splitting a given page is as follows:

1) On page allocation (either program startup or first use of the page), the page that needs to be split is duplicated. This produces two copies of the page in physical memory. We choose one page to be the target of instruction fetches, and the other to be the target of data accesses.
2) The page table entry (PTE) corresponding to the page we are splitting is set to ensure a page fault will occur on a TLB miss. In this case, the page is considered *restricted*, meaning it is only accessible when the processor is in supervisor mode. We accomplish it by setting or enabling the *supervisor* bit [29] in the PTE for that page. If *supervisor* is marked in a PTE and a user-level process attempts to access that page for any reason, a page fault will be generated and the page fault handler will be automatically invoked.
3) Depending on the reasons for the page fault, i.e., either this page fault is caused by a data TLB miss or it is caused by an instruction TLB miss, the page fault handler behaves differently. Note that for an instruction-TLB miss, the faulting address (saved in the *CR2* register) is equal to the program counter (contained in the EIP register); while for a data-TLB miss, the page fault address is different from the program counter. In the following, we describe how different TLB misses are handled. The algorithm is outlined in Algorithm 4.2.

The data-TLB is loaded from within the page fault handler. The page table entry (PTE) in question is set to point to the data page for that address, the entry is

Input: Faulting Address (addr), CPU instruction pointer (EIP), Page table Entry for addr (pte)

```
1  if addr == EIP then                                  /* Code Access */
2      pte = the_code_page;
3      unrestrict(pte);
4      enable_single_step();
5      return;
6  else                                                 /* Data Access */
7      pte = the_data_page;
8      unrestrict(pte);
9      read_byte(addr);
10     restrict(pte);
11     return;
12 end
```

Figure 4.2. SMA page fault handler

Input: Page table Entry for previously faulting address (pte)

```
1  if processor is in single step mode then
2      restrict(pte);
3      disable_single_step();
4  end
```

Figure 4.3. Debug interrupt handler

unrestricted (we unset the *supervisor* bit in the PTE), and a read of data on that page is performed. As soon as the read occurs, the memory management unit in the hardware reads the newly modified PTE, loads it into the data-TLB, and returns the content. At this point the data-TLB contains the entry to the data page for that particular address while the instruction-TLB remains untouched. Finally, the PTE is restricted again to prevent a later instruction access from improperly filling the instruction-TLB. Note that even though the PTE is restricted, later data accesses to that page can occur unhindered because the data-TLB contains a valid mapping. This loading method is also used in the PaX [31] protection model and is known to cause the overhead for a data-TLB load to be less than 2.7% in benchmarks on a Pentium III [72].

The procedure above can be seen in lines 7–11 of Algorithm 4.2. First, the page table entry is set to point to the data page and unrestricted by setting the entry to be user accessible instead of supervisor accessible. Next, a byte on the page is touched, causing the hardware to load the data-TLB with a page table entry corresponding to the data page. Finally, the page table entry is re-protected by setting it into supervisor mode once again.

The loading of the instruction-TLB has additional complications compared to that of the data-TLB, namely because there does not appear to be an equally simple procedure that can accomplish the same task. Despite these complications, however, a technique introduced in [73] can be used to load the instruction-TLB on the x86.

Once it is determined that the instruction-TLB needs to be loaded, the PTE is unrestricted, the processor is placed into single step mode, and the faulting instruction is restarted. When the instruction runs this time the PTE is read out of the page table and stored in the instruction-TLB. After the instruction finishes then the single step mode of the processor generates an interrupt, which is used as an opportunity to restrict the PTE.

This functionality can be seen in Algorithm 4.2 lines 2–5 as well as in Algorithm 4.3. First, the PTE is set to point to the corresponding code page and is

unprotected. Next, the processor is placed into single step mode and the page fault handler returns, resulting in the faulting instruction being restarted. Once the single step interrupt occurs, Algorithm 4.3 is run, effectively restricting the PTE and disabling single step mode.

We created another instruction-TLB loading method that did not require the use of single-step mode by adding a `ret` instruction to the page and then `call`ing it from the page fault handler, but surprisingly this decreased the system's efficiency. It is our theory that the slowdown was caused by the x86 maintaining cache coherency. In essence, when the write to the code page occurs, the processor invalidates the memory caches corresponding to that page, and also invalidates any portions of the instruction pipeline currently containing instructions fetched from that page. This causes undesirable performance degradation to the system.

4.4.3 Portability to Other Architectures

The TLB loading methods just described are specific for the Intel x86. In some other architecture platforms, such as SPARC, the TLB is managed by software instead of by hardware. Given this, the split memory scheme should be much easier to build. On an architecture with software loaded TLBs, there would be no need for complex data or instruction TLB loading techniques. Instead, the processor's TLBs could be loaded directly. The basic procedure would be as follows: 1) Split and mark pages and page table entries in the same way as the x86 implementation. 2) When a "memory splitting" page fault occurs, use the architecture's TLB loading instructions to load the correct TLB with an entry to the correct physical page. A project which splits memory pages to defeat software self-checksumming [73] has previously been implemented on the SPARC architecture.

Given that no complex loading procedures would be required, we believe that the code base needed to construct the SMA on such an architecture would be smaller and that the performance overhead imposed on such a system would be noticeably lower.

4.4.4 Overhead

The technique of constructing an SMA does not come without a cost. There is some overhead associated with the methodologies described above.

One potential problem is the use of the processor's single step mode for the instruction-TLB load. This loading process has a fairly significant overhead because two interrupts (the page fault and the debug interrupt) are required to complete it. This overhead ends up being minimal overall for many applications because instruction-TLB loads are fairly infrequent, as it only needs to be done *once* per page of instructions. (One TLB entry corresponds to an entire page of instructions.)

Another problem is that of context switches in the operating system. Whenever a context switch (meaning the OS changes running processes) occurs, the TLB is flushed. This means that every time a protected process is switched out and then back in, any memory accesses it makes will trigger a page fault and subsequent TLB load. The overheard of these TLB loads is significantly higher than a traditional page fault. The problem of context switches is the greatest cause of overhead in the implemented system. The experimental details of the overhead can be seen in Section 4.7.

4.4.5 Attack Response Modes

As described, the constructed SMA provides protection against the execution of injected code. We can also take advantage of the provided protection as a means of detecting the injected code execution attempt and responding accordingly. The attack is detected right before executing the first instruction injected by the attacker, therefore we can develop a number of options to respond. These include terminating the execution of the exploited process or permitting the attack to proceed while allowing its subsequent behavior to be closely monitored, similar to the way a honeypot is monitored. In the following, we describe three response modes.

Break mode

This response mode will take no action and still route the instruction fetch to the un-compromised code page, which contains null content (a string of zeroes). When it encounters zeroes as instructions, the x86 triggers a fault. As a result, the operating system will typically terminate the offending application. Notice that this option, or one that achieves the same results, is the de-facto standard for many code injection prevention systems.

Observe mode

This mode will log the code injection attempt and then still permit the attack to continue. This can be applied to honeypot-style systems wherein notification of a previously unknown attack would be helpful while still allowing the attack to continue. The system could even be tightly integrated with honeypot monitoring tools (such as Sebek [74] and VMscope [75]) to allow features such as an incoming attack being seamlessly transferred to a sandbox system and allowed to continue.

To intervene prior to the execution of injected code some sort of trap will need to be generated by the hardware. This challenge arises because the operating system does not normally intercede before every instruction fetch, and doing so would cause undue performance penalties. In our system, we take the following approach to cause a trap that will be handled by the operating system: Fill the previously empty code pages with invalid opcodes so that an invalid instruction fault will be generated when an execution attempt occurs.

Upon the detection of an invalid instruction fault, our response will be activated and Algorithm 4.4 will be executed. It works as follows: Once the trap is intercepted, log the attack attempt and record the timestamp when the injected attack code is executed. Next, the page table entry is updated to point to the data page (the data page contains the actual attack code), memory splitting is turned off for the page, the TLB entry is invalidated, and the program is resumed. The net result is that

```
Input: CPU instruction pointer (EIP), Page table entry for EIP (pte)
1 if Invalid Instruction Fault then
2     log();
3     pte = the_data_page;
4     disable_splitting(pte);
5     invalidate_tlb(pte);
6     continue_execution;
7 end
```

Figure 4.4. Observe algorithm

the PTE has been updated to point to the page containing the attack code and the attack is able to continue unhindered by the intercession.

Forensic mode

In this mode, we perform forensic analysis of the detected attack. As the attack is detected right before the first injected attack code is executed, we consider it an opportune time to start forensic analysis. Given that the OS has access to the process' entire address space as well as the current instruction pointer before malicious code is executed, forensic investigation of the attack is feasible. Operations such as shellcode analysis (the instruction pointer points to shellcode in the data pages) or attack fingerprinting based on memory contents are fully realizable and can be initiated live during a previously unseen attack. A related project – Argos [76] – has offered the ability to replace injected code with its own, "forensic" code. This same technique could easily be accommodated by this system by simply injecting the code into the process' address space, changing the EIP to point to it, and resuming program execution. In our current implementation, we dump the corresponding EIP content and the related injected attack code. An example will be presented in Section 4.5.

4.4.6 Dynamic and Shared Libraries

The concept of multiple processes sharing the same procedures and data in memory existed in the MULTICS operating system [77] as the concept of intersegment linking and addressing. Under Linux, these features are referred to as dynamic and shared libraries. For ease of presentation, we make the distinction between dynamic and shared libraries as follows: A dynamic library (sometimes called a plugin by applications) is a piece of code and data that is loaded into an application on demand at runtime while shared library is typically loaded into a process' memory space at load time. The split memory system detects the loading of these libraries at either load time or run time and splits their pages appropriately.

For libraries to be handled in a secure way they must be validated when being loaded. As a solution to this problem, we look to existing work [78, 79] that uses cryptographic primitives to verify binaries and libraries. Using one of these systems, memory splitting could simply validate the signature of the loaded library prior to loading and splitting it. This would prevent an attacker from loading a new or modified module into a running program's address space, while still permitting valid modules to be loaded and used unhindered. Given that this technique has already been implemented for two operating systems which can execute on our model of computation (Linux [78] and NetBSD [79]), we do not repeat it in our implementation.

4.5 Implementation

An x86 implementation of the above design has been created by modifying version 2.6.13 of the Linux kernel. In this section, we present a detailed description of the modifications to create the SMA.

4.5.1 Modifications to the ELF Loader

ELF is a format that defines the layout of an executable file stored on disk. The ELF loader is used to load those files into memory and begin executing them. This work includes setting up all of the code, data, BSS, stack, and heap pages as well as mapping most of the dynamic libraries used by a given program.

The modifications to the loader are as follows: After the ELF loader maps the code and data pages from the ELF file, for each one of those pages two new, side-by-side, physical pages are created and the original page is copied into both of them. This effectively creates two copies of the program's memory space in physical memory. The page table entries corresponding to the code and data pages are changed to map to one of those copies of the memory space, leaving the other copy unused for the moment. In addition, the page table entries for those pages get the supervisor bit cleared, placing that page in supervisor mode to be sure a page fault will occur when that entry is needed. A previously unused bit in the page table entry is used to signify that the page is being split. In total, about 90 lines of code are added to the ELF loader.

In this particular implementation of an SMA the memory usage of an application is effectively doubled, however this limitation is *not* one of the technique itself, but instead of the prototype. A system can be envisioned based on demand-paging (only allocating a code or data page when needed) instead of the current method of proactively duplicating every virtual page. This would result in a lower memory overhead because duplicate physical pages would only be needed when both code and data are accessed from the same virtual page.

4.5.2 Modifications to the Page Fault Handler

Under Linux, the page fault (PF) handler is called in response to a hardware generated PF interrupt. The handler is responsible for determining what caused the fault, correcting the problem, and restarting the faulting instruction.

If it is determined that the fault was caused by a split memory page and that it needs to be serviced, then the instruction pointer is compared to the faulting address to decide whether the instruction-TLB or data-TLB needs to be loaded. If the data-TLB needs to be loaded, then the PTE is set to user mode, a byte on the page is touched, and the PTE is set back to supervisor mode[1]. In the event the instruction-TLB needs to be loaded, the PTE is set to user mode (to allow access to the page) and the trap flag (single-step mode) bit in the EFLAGS register is set. This will ensure that the debug interrupt handler gets called after the instruction is restarted. Before the PF handler returns and that interrupt occurs, however, the faulting address is saved into the process' entry in the OS process table to pass it to the debug interrupt handler.

In total there were about 110 lines of code added to the PF handler to facilitate splitting memory.

4.5.3 Modifications to the Debug Interrupt Handler

The debug interrupt handler is used by the kernel to handle interrupts related to debugging. For example, using a debugger to step through a running program or watch a particular memory location makes use of this interrupt handler. For the purposes of split memory, the handler is modified to check the process table to see if a faulting address has been given, indicating that this interrupt was generated because the PF handler set the trap flag. If this is the case, then it is safe to assume that the instruction which originally caused the PF has been restarted and successfully executed (meaning the instruction-TLB has been filled) and as such the PTE is set to supervisor mode once again and the trap flag is cleared. In total, about 40 lines of code were added to the debug interrupt handler to accommodate these changes.

[1]Occasionally this procedure does not successfully load the data-TLB. In this case, single stepping mode (like the instruction-TLB load) must be used.

4.5.4 Modifications to the Memory Management System

There are a number of features related to memory management that must be slightly modified to properly handle our system. First, on program termination any split pages must be freed specially to ensure that both physical pages (the code page and data page) get put back into the kernel's pool of free memory pages. This is accomplished by simply looking for the split memory PTE bit that was set by the ELF loader, and if it is found then freeing two pages instead of one.

Another feature in the memory system that needs to be updated is the copy-on-write (COW) mechanism. COW is used by Linux to make `fork`ed processes run more efficiently. The basic idea is that when a process makes a copy of itself using `fork` both processes get a copy of the original page table, but with every entry set read-only. Then, if either process writes to a given page, the kernel will give that process its own copy. (This reduces memory usage in the system because multiple processes can share the same physical page.) For split memory the COW system must copy *both* pages in the event of a write, instead of one.

A update similar to the COW update is also made to the demand paging system. Demand paging basically means that a page is not allocated until it is required by a process. In this way a process can have a large amount of available memory space (such as in the BSS or heap) but only have physical pages allocated for portions it uses. The demand paging system was modified to allocate two pages instead of the one page it normally does. This required modifications to the code that allocates empty pages on demand as well as the code that allocates pages for memory mapped files. Proper support of memory mapped files also allows the system to protect dynamic and shared libraries as well.

Overall, about 75 lines of code were added to handle these various parts related to memory management.

4.5.5 Modifications to the Signal Handler

To accommodate the three response modes outlined in Section 4.4.5, we extend the Linux signal handler to better handle the `SIGILL` (illegal instruction) signal generated by the corresponding processor exception. In the event an attack is detected, the following three response modes have been implemented to respond to the attack: break mode, observe mode, and forensic mode.

The basic control flow in implementing the response modes is as follows. Once the attack has been detected, a log entry containing the EIP of the processor prior to malicious code execution is added to the system. After that, different modes lead to different responses:

- If the system is in observe mode, the corresponding page table entry is modified to point to the data page, split memory is disabled for that page, and the program is allowed to continue. The data page is locked in as the sole mapping and program execution is resumed. This means that only the first unauthorized code execution on a given page will be logged, as future execution will occur unhindered from the data page.
- If the system is in forensic mode (a light version of what is described in our design), we first dump additional information about the attack. For example, we record the injected attack code or shellcode. The shellcode is considered the first payload executed after compromising the vulnerable program. Thanks to the unique timing of our system in detecting the attack, we can easily identify the location of the shellcode, namely those bytes starting at the EIP in the data page. We record them in the log for later analysis. Moreover, we can also inject our own "forensic" shellcode into the address space, update the EIP to point to the new code, and resume normal program execution. Currently the implementation copies the new code onto the empty code page being executed from and changes the EIP to point to the beginning of the page. The features

of the forensic shellcode can range from a basic program exit to more advanced and customized code that collects run-time application semantic information.

- If the system is in break mode, the application will simply be terminated. This is what would occur if no modifications were made to the signal handler, and while it lacks elegance it is effective at preventing the attacker from executing his malicious code.

Overall, about 70 lines of code were added to handle these various parts related to signal handling for response mode implementation.

4.6 Effectiveness

To evaluate the effectiveness, we used a buffer overflow benchmark as well as 5 representative, real-world attacks to see how our system performs. Our testbed was a modest system, consisting of a Pentium III 600Mhz with 384 MB of RAM and a 100MBit NIC.

4.6.1 Wilander Benchmark

The code injection benchmark used for evaluation was originally put forth by Wilander et al. [28]. It was chosen because it is the only benchmark of its kind that we are aware of. The benchmark was modified slightly to allow it to handle having the code injected on the data, BSS, heap, and stack portions of the program's address space. In addition, four of the test cases did not successfully execute an attack on our unprotected system, and so have been labeled "N/A." Table 4.1 shows the results of running the benchmark. The checkmarks indicate that the system successfully halted the attack. As can be seen, the system was effective in preventing all types of code injection attacks present in the benchmark. The effectiveness of the system is because no matter what method of control-flow hijacking the benchmark uses, the processor is simply unable to fetch the injected code.

Table 4.1
Wilander benchmark results when code is injected onto the data, BSS, heap, and stack segments

Attack Type	Hijack Type	Injection Destination			
		Data	BSS	Heap	Stack
Buffer overflow on stack	Return address	✓	✓	✓	✓
	Old base pointer	✓	✓	✓	✓
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/BSS	Function pointer	✓	✓	✓	✓
	Longjmp buffer	✓	✓	✓	✓
Buffer overflow of pointers on stack	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as local variable	✓	✓	✓	✓
	Function pointer as parameter	✓	✓	✓	✓
	Longjmp buffer as local variable	✓	✓	✓	✓
	Longjmp buffer as function parameter	✓	✓	✓	✓
Buffer overflow on heap/BSS	Return address	N/A	N/A	✓	N/A
	Old base pointer	N/A	N/A	N/A	N/A
	Function pointer as variable	✓	✓	✓	✓
	Longjmp buffer as variable	✓	✓	✓	✓

4.6.2 Real World Attacks

Five representative software packages containing real-world vulnerabilities that permit code injection and execution were run under our implementation. Vulnerabilities in five major Linux server packages from 2001 to 2003 were used. These specific vulnerabilities were chosen because of the availability and effectiveness of publicly released exploits. Our software platform for the attacks was a copy of the RedHat 7.2 operating system (chosen because of its vulnerability to many attacks from that time period) that had been manually upgraded to use version 2.6.13 of the Linux kernel. Table 4.2 summarizes the results of the experiments, including the versions of software installed on our testing platform. Some software shipped with the default version of RedHat 7.2, other software was "forward" ported from previous releases. The results of the attacks when executed on an unpatched kernel is reflected by the "Attack Result" column.

1. Apache 1.3.20 with OpenSSL 0.9.6d. A bug in OpenSSL allows a buffer overflow to occur if an attacker sends a large client master key to the server. The exploit we used, openssl-too-open by Solar Eclipse [80], overflows a heap buffer and makes use of an information leak in the SSL handshake to determine the proper address for its shellcode. If the attack successfully executes, a shell owned by `nobody` (the `uid` of the apache process) is spawned over the network to the attacker. When run under our system, the heap buffer is overflowed, but execution of the injected shell code is foiled because it is unavailable to the processor when it attempts to fetch instructions from the heap page.
2. Bind 8.2.2_P5. Bugs in the DNS server implementation allow either a stack or heap overflow to occur (depending on which bug is exploited) while handling transaction signatures. For our testing we used a publicly released lsd-pl.net exploit [81]. (A modified version of this same exploit code was used by the Lion worm [82].) Much like the apache attack, this exploit makes used of an information leak bug to determine the shellcode jump address. Once that occurs

Table 4.2
Five real world vulnerabilities

Software	**CVE**	**Corruptable Memory Region**	**Attack Result**	**Stopped?**
apache-1.3.20-16/mod_ssl-2.8.4-9	CVE-2002-0656	Heap	`nobody` shell	Yes
bind-8.2.2_P5-9	CVE-2001-0010	Stack or Heap	`named` shell	Yes
proftpd-1.2.7-1	CVE-2003-0831	Heap	`root` shell	Yes
samba-2.2.1a-4	CVE-2003-0201	Stack	`root` shell	Yes
wu-ftpd-2.6.1-18	CVE-2001-0550	Heap	`root` shell	Yes

a stack overflow is triggered and a shell is spawned over the network. When run under our system, the information leak bug still functions and the stack overflow still occurs, but the shellcode is unable to be fetched and the execution attempt fails.

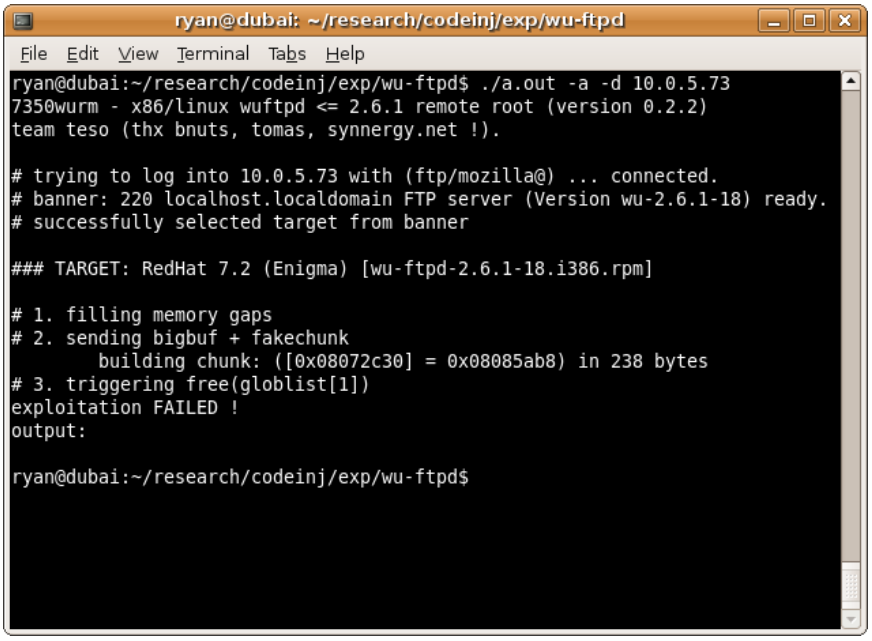
3. ProFTPD 1.2.7. When transferring files in ASCII mode, ProFTPD contains a bug that causes newline characters to be translated incorrectly and permits an attacker to execute arbitrary code. Our exploit of choice was proftpd-not-pro-enough by Solar Eclipse [83]. To trigger the flaw the exploit logs in to the server and uploads a file containing a malicious payload. Next, it puts the server in ASCII mode and downloads that file. During the ASCII translation process the exploit code is executed from the heap. The malicious code then breaks out of any `chroot` environments and spawns a root shell over the network. Executing the server under our system results in the instruction fetch from the heap failing and hence the attack is foiled.
4. Samba 2.2.1a. Samba contains a bug in the `call_trans2open` function that allows a stack buffer to be overflowed. For our testing we used an exploit put out by eSDee [84]. The exploit is a stack based buffer overflow with a brute-force mode to guess the address of the shellcode on the stack based on a good "first guess" obtained by manual analysis of a similar vulnerable system. This bug was made more difficult to exploit because version 2.6 of the Linux kernel added randomization to the placement of an application's stack within memory. This means that it can take a long time for the attack to properly determine the correct stack address. To better facilitate testing, the exploit was "helped" by providing a better first guess using insider information about the stack location. (An unmodified attack would still function given enough time.) When run under our system, the return address is still guessed properly, but the shellcode is unavailable to the processor when it attempts to transfer control to that location.

5. WU-FTPD 2.6.1. A bug in the WU-FTPD code handling filename globbing (the feature that expands strings like `*.txt` into all the `.txt` files in a directory) combined with the `free`'ing of attacker controlled memory permits arbitrary code execution. This bug is different from, but related to, a traditional heap overflow. The exploit code we used was 7350wurm published by TESO Security [85]. The exploit logs in to the server, adds its own malicious code to the heap, triggers the globbing flaw, and causes a root shell to be spawned. Under our system, the heap is still filled with malicious code and the globbing bug is still triggered, but the injected code is not fetched by the processor.

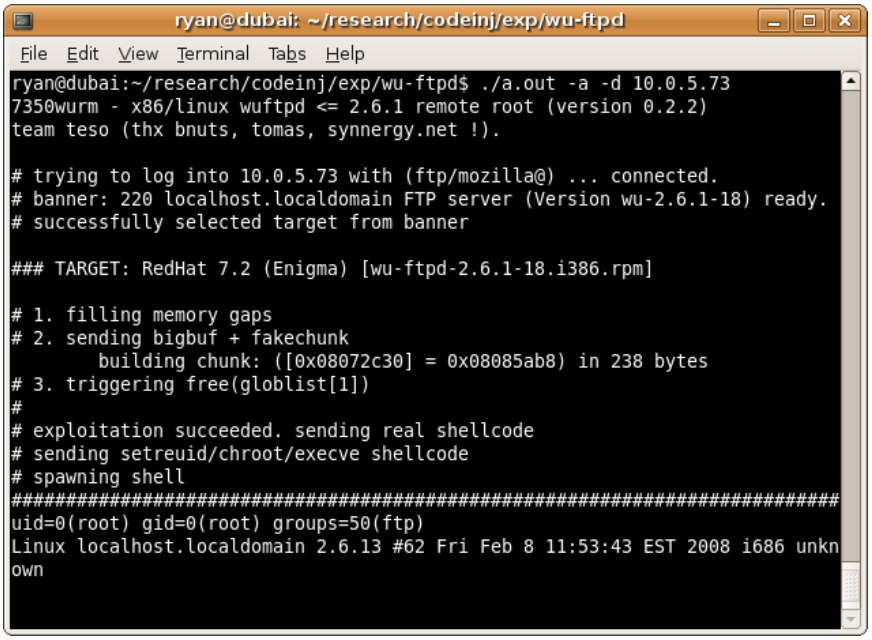
Overall, even with a variety of bugs and exploitation techniques, our system is able to defeat code injection in these real-world scenarios because it prevents malicious code from ever being executed, even after successful injection into the process' data space.

4.6.3 Response Modes

To validate the attack response modes described in Section 4.4.5, the WU-FTPD vulnerability and exploit were executed under the various modes. Figures 4.5 and 4.6 show how the exploit code reacts when the WU-FTPD daemon is run under break mode, observe mode, and forensic mode. First, the ftp server is run under break mode. As can be seen in Figure 4.5(a), the exploit fails to successfully launch a root shell. (This is because the process crashes when attempting to execute the shellcode.) This is contrasted with our second test, executing the server under observe mode, where the exploit is allowed to continue unhindered and a rootshell is spawned (Figure 4.5(b)). More information about this particular attack can be observed when running under forensic mode, which can be seen in Figure 4.6(a). A closer examination of the screenshot will find that the log entry contains the first 20 bytes of the injected shellcode. This can be recognized because of the `nop` instructions (the `0x90` bytes).

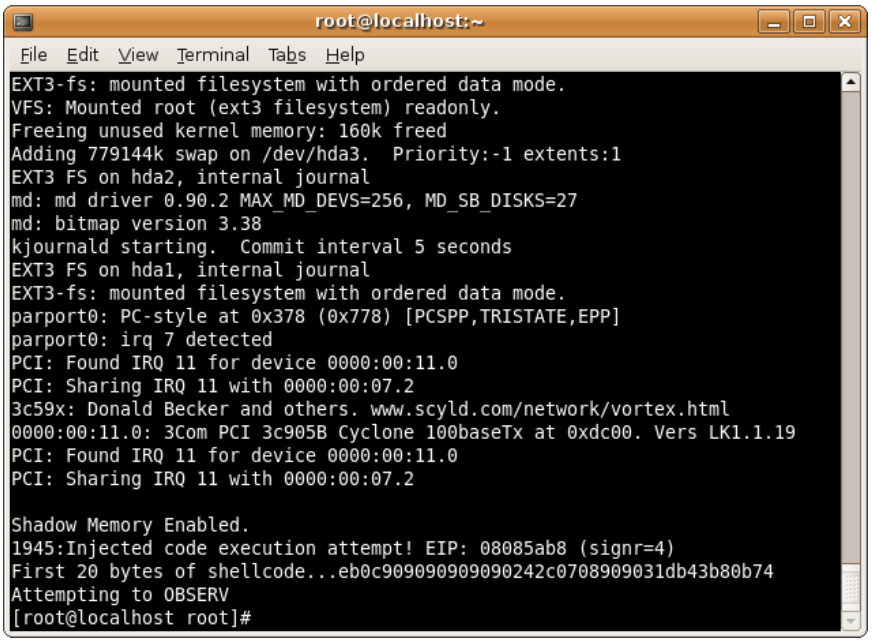


(a) Attack failure during break mode

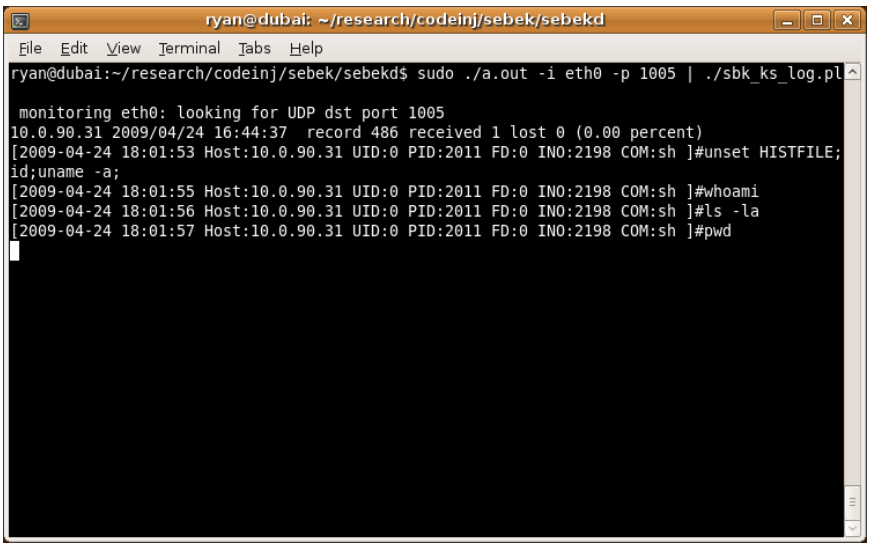


(b) Attack success during observe mode

Figure 4.5. Demonstration of response modes against the WU-FTPD exploit (Part 1)



(a) Output during forensics mode



(b) Sebek log during observe mode

Figure 4.6. Demonstration of response modes against the WU-FTPD exploit (Part 2)

A manual analysis of the exploit code reveals that these 20 bytes are indeed the first 20 bytes of injected code. The exploit functions using two stages of injected code. The initial stage (the first 20 bytes of which are in the figure) is used to write 4 bytes back to the attacker over the network to signal that the attack succeeded and then immediately reads a second stage of shellcode from the network and executes it. Currently, our system can successfully observe the execution of the initial stage of code, but does not intercede before the second stage because the memory page has been locked on to the data entry.

The last screenshot, Figure 4.6(b), demonstrates our system used in conjunction with Sebek, a kernel level logging mechanism for honeypots. In our experiment, we integrate Sebek as a part of the observe response mode. By default, Sebek's logging mechanism always runs. To reduce log volume, we modified Sebek to be activated by a buffer overflow event (caused by code injection) detected by our system. By doing so, log files can be significantly smaller, yet we can still ensure that an attacker's actions are captured thanks to our system's detection of code injection attacks. The screenshot shows Sebek logging the commands the attacker types into his spawned shellcode.

We also tested the possibility of injecting custom shellcode into the program's address space. For demonstration purposes we injected the code required to cause the program to call the `exit` system call and terminate gracefully. The injected shellcode (corresponding to `exit(0);`) is as follows:

```
"\xbb\x00\x00\x00\x00" /* mov $0x0,%ebx */
"\xb8\x01\x00\x00\x00" /* mov $0x1,%eax */
"\xcd\x80";            /* int $0x80     */
```

The code loads `%ebx` with the program's return value (`0`), loads the system call number for `exit()` into `%eax`, and finally generates the interrupt required for the system call. By replacing the attacker's injected code with this code, the program terminates *without* a segmentation fault. While this test shows the injection of fairly uninvolved

code, it can easily be replaced with more sophisticated forensic shellcode to assist in attack investigation.

4.7 Performance

A number of benchmarks, both applications and micro-benchmarks, were used to test the performance of the system. When applicable, benchmarks were run 10 times and the results averaged. Details of the configuration for the tests are available in Table 4.3. Each result has been normalized with respect to the speed of the unprotected system.

Four benchmarks that we consider to be a reasonable assessment of the system's performance can be found in Figure 4.7. They were chosen because they test a variety of both CPU and I/O intensive workloads. First, the Apache [86] webserver was run in a threading mode to serve a 32KB page (roughly the size of Purdue University's main index.html). The ApacheBench program was then run on another machine connected via the NIC to determine the request throughput of the system as a whole. The protected system achieved a little over 89% of the unprotected system's throughput. Next, gzip was used to compress a 256 MB file, and the operation was timed. The protected system was found to run at 87% of full speed. Third, the nbench [87] suite was used to show performance under a set of primarily computation based tests. The slowest test in the nbench system came in at slightly under 97%. Finally, the Unixbench [88] Unix benchmarking suite was used as a micro-benchmark to test various aspects of the system's performance at tasks such as process creation, pipe throughput, filesystem throughput, etc. Here, the split memory system ran at 82% of normal speed. This result will be explained below. As can be seen from these four benchmarks, the system performance at above 80% of full speed under a variety of tasks.

Two benchmarks contrived to highlight the system's weakness can be found in Figure 4.8. First, one of the Unixbench test cases called "pipe based context switch-

Table 4.3
Configuration information used for performance evaluation

Item	**Version**	**Configuration**
Slackware	10.2.0	Using Linux 2.6.13
Apache	2.2.3	Worker mpm mode, set to spawn one process with threads
ApacheBench	2.0.41-dev	`-c3 -t 60 <url/file>`
Unixbench	4.1.0	N/A
Nbench	2.2.2	N/A
Gzip	1.3.3	Compress a 256 MB file.

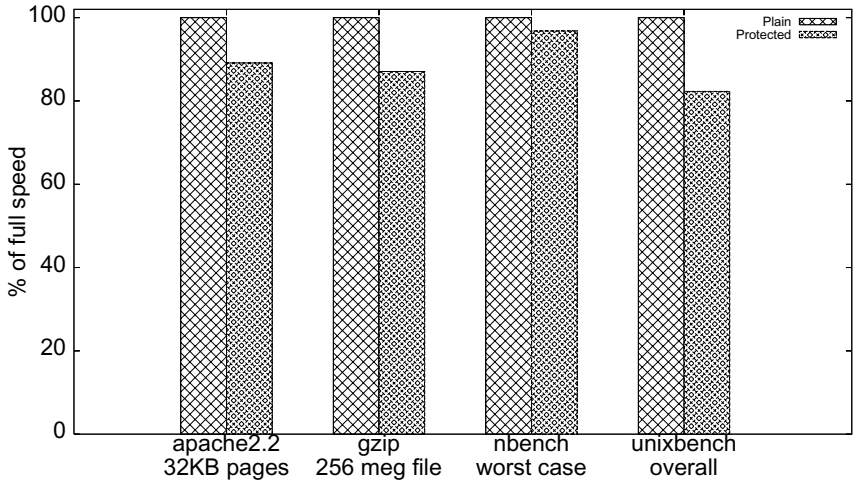


Figure 4.7. Normalized performance for applications and benchmarks

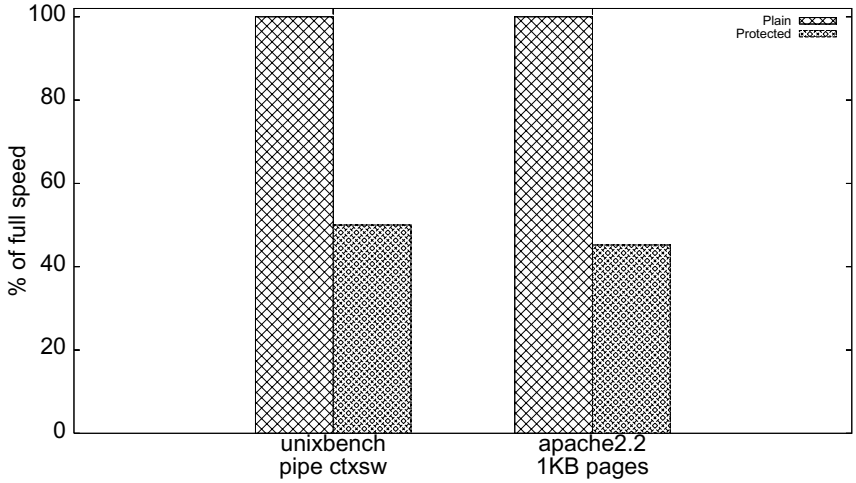


Figure 4.8. Stress-testing the performance penalties from context switching

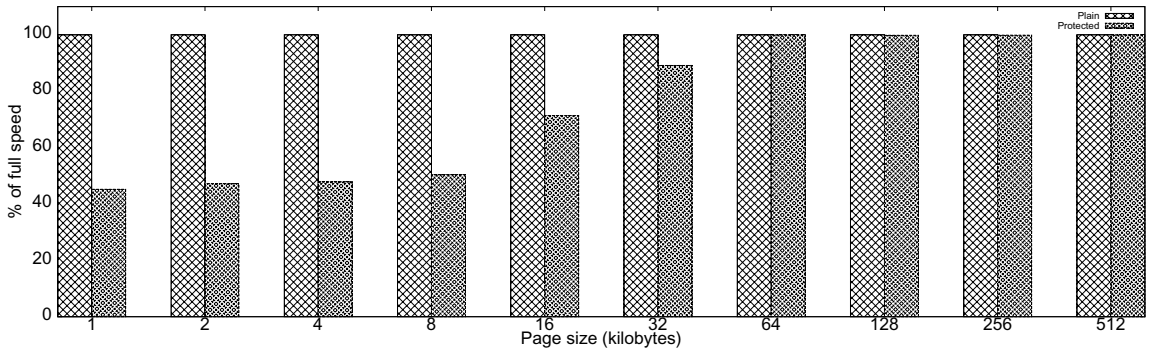


Figure 4.9. Closer look into Apache performance

ing" is shown. This primarily tests how quickly a system can context switch between two processes that are exchanging data using a pipe. The next test is Apache used to serve a 1KB page. In this configuration, Apache will context switch heavily while serving requests. In both of these tests, context switching is taken to an extreme and therefore our system's performance degrades substantially because of the constant flushing of the TLB. As can be seen in the graph, both are at or below 50%. In addition, in Figure 4.9, we have a more thorough set of Apache benchmarks demonstrating this same phenomena, namely that for low page sizes the system context switches heavily and performance suffers, whereas for larger page sizes that cause Apache to spend more time on I/O as well as begin to saturate the system's network link, the results become significantly better. These tests are indicative of the system's *worst-case* performance under highly stressful conditions.

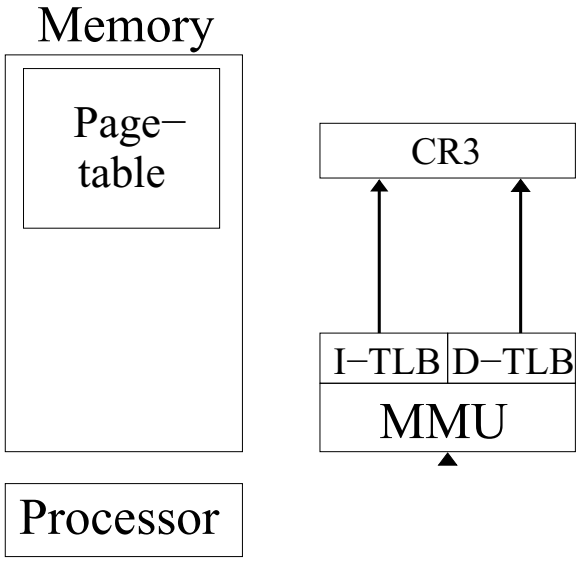
Overall, the system's performance is, in most cases, between 80 and 90% of an unprotected system. Moreover, if split memory was supported at the hardware level, the overheard would be almost non-existent.

4.8 Hardware Support

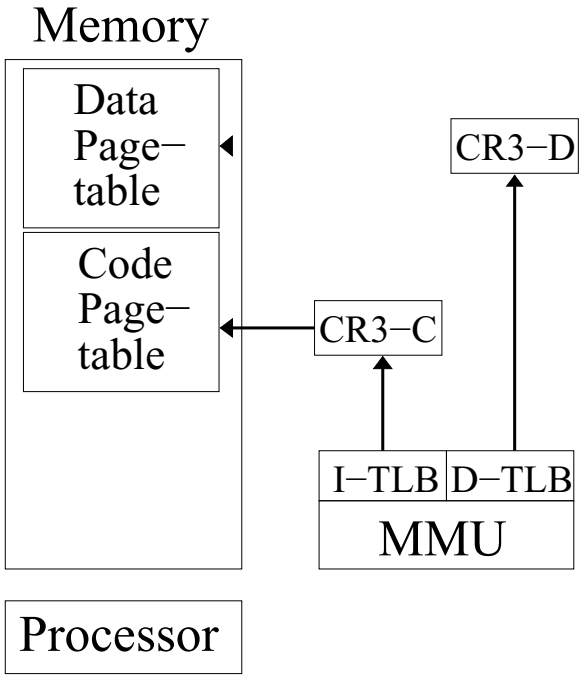
Although this chapter discusses an operating system modification to enable an SMA for user-level programs, if the feature was supported in hardware then the performance overhead described in Section 4.7 would be greatly reduced. In this section we will discuss what changes would need to be made to a basic x86 architecture to support a user-level SMA.

Figure 4.10(a) is a representation of the paging architecture of the x86 architecture. The important thing to note is that while the memory management unit (MMU) has access to two different TLBs, one for code and one for data, there is only one page table register (CR3) and only one page table.

Figure 4.10(b) represents our modifications to the paging architecture. As can be seen, there are still two TLBs, but now there are also two page table registers (CR3-C



(a) Standard x86 paging hardware



(b) Modified x86 paging hardware

Figure 4.10. Modifications to x86 to support user-level memory splitting in hardware

for code and CR3-D for data) and two page tables as well. Under this new design all instruction accesses use one page table and associated TLB and all data accesses use another. An operating system creating an SMA for a process would create two different page tables, one for code and one for data, and load CR3-C and CR3-D appropriately. This would cause the process to run inside of an SMA. This technique is also easily made backwards compatible with non-SMA processes by creating only one page table and pointing both CR3-C and CR3-D to it.

This is not the first architecture to make use of two separate address spaces for a single user-level process. The PDP 11/45 had a processor [89] with a similar design.

While other hardware modifications to achieve the same goal (such as a reworked instruction set or extended segmentation support) may also be valid, this one is presented here because it is a straightforward extension of the already discussed and tested software design.

4.9 Limitations

There are a few limitations to our approach. First, as shown in other work [90], a split memory architecture does not lend itself well to handling self-modifying code. As such, self-modifying programs cannot be protected using our technique.

Next, this protection scheme offers no protection against attacks which do not rely on executing code injected by the attacker. For example, modifying a function's return address to point to a different part of the original code pages will not be stopped by this scheme. Address space layout randomization [36] could be combined with our technique to help prevent this kind of attack. Along those same lines, non-control-data attacks [18], wherein an attacker modifies a program's data to alter program flow, are also not protected by this system.

4.10 Summary

In this chapter we have demonstrated the efficacy of the SMA for the prevention of code injection at the user-level. Instead of maintaining the traditional single memory space containing both code and data, which is often exploited by code injection attacks, our approach creates an SMA that separates code and data into different memory spaces. Consequently, in a system protected by our approach, code injection attacks may result in the injection of attack code into the data space. However, the attack code in the data space cannot be fetched for execution as instructions are only retrieved from the code space. We have implemented a Linux prototype on the x86 architecture, and experimental results show the system is effective in preventing and responding to a wide range of code injection attacks in both artificial and real-world scenarios and performs between 80 and 90% of full speed in most cases.

5 SMA FOR KERNEL-LEVEL CODE INJECTION DEFENSE

We will now demonstrate the applicability of an SMA to preventing kernel-level code injection attacks. Specifically, we will be discussing code injection based kernel rootkit attacks [91].

5.1 Introduction

In this chapter we present NICKLE ("No Instruction Creeping into Kernel Level Executed")[1], a lightweight, VMM-based system that provides an important guarantee in kernel rootkit prevention: *No unauthorized code can be executed at the kernel level.* NICKLE achieves this guarantee using legacy hardware and without requiring guest OS kernel modification. As such, NICKLE is readily deployable to protect unmodified guest OSes (e.g., Fedora Core 3/4/5 and Windows 2K/XP) against kernel rootkits. NICKLE is based on observing a common, fundamental characteristic of most modern kernel rootkits: their ability to execute unauthorized instructions at the kernel level.

To achieve the "NICKLE" guarantee, we first observe that a kernel rootkit is able to access the entire physical address space of the victim machine. This observation inspires us to impose restricted access to the instructions in the kernel space: only *authenticated* kernel instructions can be fetched for execution. Obviously, such a restriction cannot be enforced by the OS kernel itself. Instead, a natural strategy is to enforce such a memory access restriction using the VMM, which is at a privilege level higher than that of the (guest) OS kernel.

Our main challenge is to realize the above VMM-level kernel instruction fetch restriction in a guest-transparent, real-time, and efficient manner. An intuitive ap-

[1]With a slight abuse of terms, we use NICKLE to denote both the system itself and the guarantee achieved by the system – when used in quotation marks.

proach would be to impose W⊕X on kernel memory pages to protect existing kernel code and prevent the execution of injected kernel code. However, because of the existence of mixed kernel pages in commodity OSes, this approach is not viable for guest-transparent protection. To address that, we propose a VMM-based SMA for NICKLE that will work with mixed kernel pages. An SMA for kernel level code injection prevention makes use of two memory spaces: one for kernel code and the other for everything else. We refer to the kernel code memory space as the *shadow memory* and the memory space for everything else as the *standard memory*. The VMM enforces that the guest OS kernel cannot access the shadow memory. Upon the VM's startup, the VMM performs kernel code authentication and dynamically copies authenticated kernel instructions from the standard memory to the shadow memory. At runtime, any instruction executed in the kernel space must be fetched from the shadow memory instead of from the standard memory. To enforce this while maintaining guest transparency, a lightweight *guest memory access indirection* mechanism is added to the VMM. As such, a kernel rootkit will never be able to execute any of its own code as the code injected into the kernel space will not be able to reach the shadow memory.

We have implemented NICKLE in two VMMs: QEMU [92] with the KQEMU accelerator and VirtualBox [93]. Our evaluation results show that NICKLE incurs a reasonable impact on the VMM platform (e.g., 1.01% on QEMU+KQEMU and 5.45% on VirtualBox when running Unixbench). NICKLE is shown capable of transparently protecting a variety of commodity OSes, including RedHat 8.0 (Linux 2.4.18 kernel), Fedora Core 3 (Linux 2.6.15 kernel), Windows 2000, and Windows XP. Our results show that NICKLE is able to prevent and respond to 22 real-world kernel rootkits targeting the above OSes, without requiring details of rootkit attack vectors. Finally, our porting experience indicates that the NICKLE design is generic and realizable in a variety of VMMs.

5.2 NICKLE Design

5.2.1 Design Goals and Threat Model

NICKLE has the following three main design goals:

First, NICKLE should prevent any unauthorized code from being executed in the kernel space of the protected VM. The challenges of realizing this goal come from the real-time requirement of prevention as well as from the requirement that the guest OS kernel should not be trusted to initiate any task of the prevention – the latter requirement is justified by the kernel rootkit's highest privilege level inside the VM and the possible existence of zero-day vulnerabilities inside the guest OS kernel. NICKLE overcomes these challenges using a VMM-based SMA (Section 5.2.2). We note that the scope of NICKLE is focused on preventing unauthorized kernel code execution. The prevention of other types of attacks (e.g., data-only attacks) is a non-goal and related solutions will be discussed in Section 5.5.

Second, NICKLE should not require modifications to the guest OS kernel. This allows commodity OSes to be supported "as is" without recompilation and reinstallation. Correspondingly, the challenge in realizing this goal is to make the SMA transparent to the VM with respect to both the VM's function and performance.

Third, the design of NICKLE should be generically portable to a range of VMMs. Given this, the challenge is to ensure that NICKLE has a small footprint within the VMM and remains lightweight with respect to performance impact. We focus on supporting NICKLE in software VMMs, but we expect that the exploitation of recent hardware-based virtualization extensions [94,95], will improve NICKLE's performance even further.

In addition, it is also desirable that NICKLE facilitate various flexible response mechanisms to be activated upon the detection of an unauthorized kernel code execution attempt. A flexible response, for example, is to cause only the offending process to fail without impacting the rest of the OS. The challenge in realizing this is to

initiate flexible responses entirely from outside the protected VM and minimize the side-effects on the running OS.

We assume the following adversary model when designing NICKLE: (1) The kernel rootkit has the highest privilege level inside the victim VM (e.g., *root* privileges in a UNIX system); (2) The kernel rootkit has full access to the VM's memory space (e.g., through `/dev/kmem` in Linux); (3) The rootkit needs to execute its own (malicious) code in the kernel space. Such a need exists in most kernel rootkits today [47], and we will discuss possible exceptions in Section 5.5.

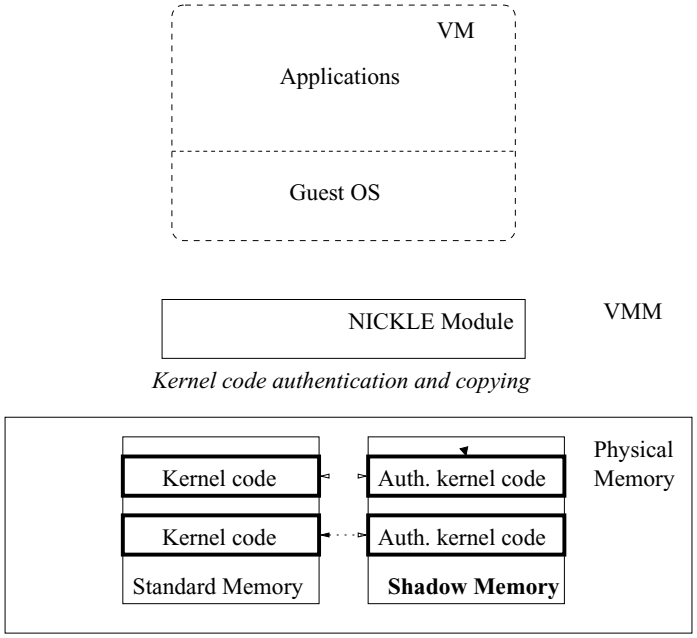
Meanwhile, we assume a trusted VMM that provides VM isolation. This assumption is shared by many other VMM-based security research efforts [49, 50, 75, 96–98]. We will discuss possible attacks (e.g., VM fingerprinting) in Section 5.5. With this assumption, we consider the threat from DMA attacks launched from physical hosts outside of the scope of this work.[2]

5.2.2 VMM-based SMA

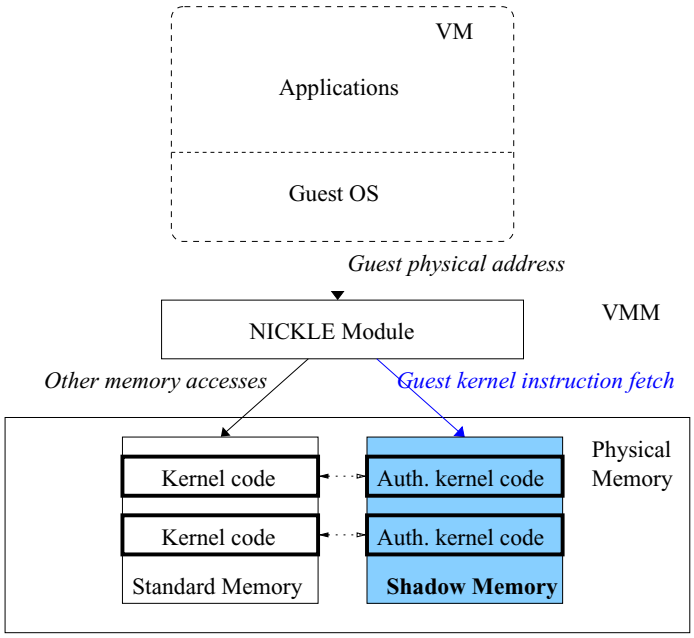
The VMM-based SMA enforces the "NICKLE" property as follows. For a VM, apart from its standard physical memory space, the VMM also allocates a separate physical memory region as the VM's *shadow memory* which is transparent to the VM and controlled by the VMM. Upon the startup of the VM's OS, all known-good, authenticated guest kernel instructions will be copied from the VM's standard memory to the shadow memory (Figure 5.1(a)). At runtime, when the VM is about to execute a kernel instruction, the VMM will transparently redirect the kernel instruction fetch to the shadow memory (Figure 5.1(b)). All other memory accesses (to user code, user data, and kernel data) will proceed unhindered in the standard memory.

The design of the VMM-based SMA is motivated by the observation that modern computers define a single memory space for all code, both kernel code and user code,

[2]There exists another type of DMA attack that is initiated from within a guest VM. However, since the VMM itself virtualizes or mediates the guest DMA operations, NICKLE can be easily extended to intercede and block them.



(a) Kernel code authorization and copying



(b) Guest physical address redirection

Figure 5.1. VMM-based SMA in NICKLE

and data. With the VMM running at a higher privilege level, we can now "shadow" the guest kernel code space with elevated (VMM-level) privileges to ensure that the guest OS kernel itself cannot access the shadowed kernel code space containing the authenticated kernel instructions. By doing so, even if a kernel rootkit is able to inject its own code into the VM's standard memory, the VMM will ensure that the malicious code never gets copied over to the shadow memory. Moreover, an attempt to execute the malicious code can be caught immediately because of the inconsistency between the standard and shadow memory contents.

An important question to answer is, "How is NICKLE functionally different from W⊕X?" In essence, W⊕X is a scheme that enforces the property, "A given memory page will never be both writable and executable at the same time." The basic premise behind this scheme is that if a page cannot be written to and later executed from, code injection becomes impossible. There are two main reasons why this scheme is not adequate for stopping kernel level rootkits:

First, W⊕X is not able to protect mixed kernel pages with both code and data, which do exist in some OSes. As a specific example, in a Fedora Core 3 VM (with the 32-bit 2.6.15 kernel and the NX protection), the Linux kernel stores the main static kernel text in memory range $[0xc0100000, 0xc02dea50]$ and keeps the system call table starting from virtual address $0xc02e04a0$. Notice that the Linux kernel uses a large page size ($2MB$) to manage the physical memory,[3] which means that the first two kernel pages cover memory ranges $[0xc0000000, 0xc0200000)$ and $[0xc0200000, 0xc0400000)$, respectively. As a result, the second kernel page contains both code and data, and thus must be marked both writable and executable – This conflicts with the W⊕X scheme. Mixed pages also exist for accommodating the code and data of Linux loadable kernel modules (LKMs) – an example will be shown in Section 5.4.1. NICKLE is able to protect mixed pages.[4]

[3]If the NX protection is disabled, those kernel pages containing static kernel text will be of $4MB$ in size.

[4]We also considered the option of eliminating mixed kernel pages. However, doing so would require kernel source code modification, which conflicts with our second design goal. Even given source code access, mixed page elimination is still a complex task (more than only page-aligning data). A

Second, W⊕X assumes only one execution privilege level while kernel rootkit prevention requires further distinction between user and kernel code pages. For example, a page may need to be set executable in user mode but non-executable in kernel mode. The sort of permission desired is not W⊕X, but W⊕KX (i.e. not writable and kernel-executable at the same time.) Still, we point out that the enforcement of W⊕KX is *not* effective for mixed kernel pages and, regardless, not obvious to construct on x86 processors that do not allow such fine-grained memory permissions.

5.2.3 Guest Memory Access Indirection

To construct the VMM-based SMA, two issues need to be resolved. The first is adding authenticated kernel code to the shadow memory. The second is fetching authenticated kernel instructions for execution while detecting and preventing any attempt to execute unauthorized code in the kernel space. Our solutions need to be transparent to the guest OS (and thus to the kernel rootkits). We now present the guest memory access indirection technique to address these issues.

Guest memory access indirection is performed between the VM and its memory (standard and shadow) by a thin NICKLE module inside the VMM. It has two main functions, kernel code authentication and copying at VM startup and upon kernel module loading as well as guest physical address redirection at runtime.

In some ways this component of NICKLE's design can be thought of as a reference monitor [99] which validates accesses (memory reads and writes) based on a set of permissions (the processor's current privilege level and type of access).

To add authenticated kernel instructions to the shadow memory, the NICKLE module inside the VMM needs to first determine the accurate timing for kernel code authentication and copying. To better articulate the problem, we will use the Linux kernel as an example. There are two specific situations throughout the OS's lifetime

kernel configuration option with a similar purpose exists in the latest Linux kernel (version 2.6.23). But after we enabled the option, we still found more than 700 pages that were both writable and executable. NICKLE instead simply avoids such complexity and works even with mixed kernel pages.

when kernel code needs to be authorized and shadowed: One at startup and the other upon the loading/unloading of a loadable kernel module (LKM). When the VM is booting, the guest's shadow memory is empty. The kernel bootstrap code then decompresses the kernel. Right after the decompression and before any processes are executed, NICKLE will use a cryptographic hash to verify the integrity of the kernel code (this is very similar to level 4 in the secure bootstrap procedure [100]) and then copy the authenticated kernel code from the standard memory into the shadow memory (Figure 5.1(a)). As such, the protected VM will start with a known clean kernel.

The LKM support in modern OSes complicates our design. From NICKLE's perspective, LKMs are considered injected kernel code and thus need to be authenticated and shadowed before their execution. The challenge for NICKLE is to *externally* monitor the guest OS and detect the kernel module loading/unloading events in real-time. NICKLE achieves this by leveraging previous work on non-intrusive VM monitoring and semantic event reconstruction [49, 75]. When NICKLE detects the loading of a new kernel module, it intercepts the VM's execution and performs kernel module code authentication and shadowing. The authentication is performed by taking a cryptographic hash of the kernel module's code segment and comparing it with a known correct value, which is computed a priori off-line and provided by the administrator or distribution maintainer.[5] If the hash values do not match, the kernel module's code will not be copied to the shadow memory. This technique is similar is principle and goal to integrity shells [54, 55].

Through kernel code authentication and copying, only authenticated kernel code will be loaded into the shadow memory, thus blocking the copying of malicious kernel rootkit code or any other code injected by exploiting kernel vulnerabilities, including zero-day vulnerabilities. It is important to note that neither kernel startup hashing nor kernel module hashing assumes trust in the guest OS. Should the guest OS fail

[5]We have developed an off-line kernel module profiler that, given a legitimate kernel module, will compute the corresponding hash value (Section 5.3.1).

to cooperate, *no* code will be copied to the shadow memory, and any attempts to execute that code will be detected and refused.

Once the shadow memory contains the authenticated kernel code, the operating system memory accesses must be redirected at runtime. The NICKLE module inside the VMM intercepts the memory accesses of the VM *after* the "guest virtual address → guest physical address" translation. As such, NICKLE does not interfere with – and is therefore transparent to – the guest OS's memory access handling procedure and virtual memory mappings. Instead, it takes the guest physical address, determines the type of the memory access (kernel, user; code, data; etc.), and routes it to either the standard or shadow memory (Figure 5.1(b)).

We point out that the interception of VM memory accesses can be provided by existing VMMs (e.g., QEMU+KQEMU, VirtualBox, and VMware). NICKLE builds on this interception capability by adding the guest physical address redirection logic. First, using a simple method to check the current privilege level of the processor, NICKLE determines whether the current instruction fetch is for kernel code or for user code: If the processor is in supervisor mode (CPL=0 on x86), we infer that the fetch is for kernel code and NICKLE will verify and route the instruction fetch to the shadow memory. Otherwise, the processor is in user mode and NICKLE will route the instruction fetch to the standard memory. Data accesses of either type are always routed to the standard memory.

5.2.4 Flexible Responses to Unauthorized Kernel Code Execution Attempts

If an unauthorized execution attempt is detected, there are a number of ways NICKLE can respond. Given that NICKLE is situated between the VM and its memory and has a higher privilege level than the guest OS, it possesses a wide range of options and capabilities to respond. We describe two response modes facilitated by the current NICKLE system.

Rewrite mode: NICKLE will dynamically rewrite the malicious kernel code with code of its own. The response code can range from OS-specific error handling code to a well-crafted payload designed to clean up the impact of a rootkit installation attempt. Note that this mode may require an understanding of the guest OS to ensure that valid, sensible code is returned.

Break mode: NICKLE will take no action and route the instruction fetch to the *shadow memory*. In the case where the attacker only modifies the original kernel code, this mode will lead to the execution of the original code – a desirable situation. However, in the case where *new* code is injected into the kernel, this mode will lead to an instruction fetch from presumably null content (containing 0s) in the shadow memory. As such, break mode prevents malicious kernel code execution but may or may not be graceful depending on how the OS handles invalid code execution faults.

5.3 NICKLE Implementation

To validate the portability of the NICKLE design, we have implemented NICKLE in two VMMs: QEMU+KQEMU [92] and VirtualBox [93]. As the open-source QEMU+KQEMU is the VMM platform where we first implemented NICKLE, we use it as the representative VMM to describe our implementation details. For most of this section, we choose RedHat 8.0 as the default guest OS. We will also discuss the limitations of our current prototype in supporting Windows guest OSes.

5.3.1 Memory Shadowing and Guest Memory Access Indirection

To implement memory shadowing, we have considered two options: (1) NICKLE could interfere as instructions are executed; or (2) NICKLE could interfere when instructions are dynamically translated. Note that dynamic instruction translation is a key technique behind existing software-based VMMs, which transparently translates guest machine code into native code that will run on the physical host. We favor the second option for performance reasons: QEMU caches translated code blocks, and

NICKLE can take advantage of this. In QEMU+KQEMU, for example, guest kernel instructions are grouped into "blocks" and are dynamically translated at runtime. After a block of code is translated, it is stored in a cache to make it available for future execution. In terms of NICKLE, this means that if we intercede during code translation we need not intercede as often as we would if we did so during code execution, resulting in a smaller impact on system performance.

The pseudo-code for memory shadowing and guest memory access indirection is shown in Algorithm 5.2. Given the guest physical address of an instruction to be executed by the VM, NICKLE first checks the current privilege level of the processor (CPL) to determine if it is in supervisor mode. Using the guest physical address, NICKLE compares the content of the standard and shadow memories to determine whether the kernel instruction to be executed is already in the shadow memory (namely has been authenticated). If so, the kernel instruction is allowed to be fetched, translated, and executed. If not, NICKLE will determine if the guest OS kernel is being bootstrapped or a kernel module is being loaded. If either is the case, the corresponding kernel text or kernel module code will be authenticated and, if successful, shadowed into the shadow memory. Otherwise, NICKLE detects an attempt to execute an unauthorized instruction in the kernel space and prevents it by executing our response to the attempt.

In Algorithm 5.2, the way to determine whether the guest OS kernel is being bootstrapped or a kernel module is being loaded requires OS-specific knowledge. Using the Linux 2.4 kernel as an example, when the kernel's *startup_32* function, located at physical address `0x00100000` or virtual address `0xc0100000` as shown in the *System.map* file, is to be executed, we know that this is the first instruction executed to load the kernel and we can intercede appropriately. For kernel module loading the NICKLE module inside the VMM can intercept the system call used to load modules and then perform kernel module authentication and shadowing right before the module-specific *init_module* routine is executed.

Input: (1) GuestPA: guest physical address of instruction to be executed; (2) ShadowMEM[]: shadow memory; (3) StandardMEM[]: standard memory

```
1 if !IsUserMode(vcpu) AND ShadowMEM[GuestPA] != StandardMEM[GuestPA] then
2     if (kernel is being bootstrapped) OR (module is being loaded) then
3         Authenticate and shadow code;
4     else
5         Unauthorized execution attempt - Execute response;
6     end
7 end
8 Fetch, translate, and cache code;
```

Figure 5.2. Algorithm for memory shadowing and guest memory access indirection

In our implementation, the loading of LKMs requires special handling. Providing a hash of a kernel module's code space ends up being complicated in practice. This is because kernel modules are dynamically relocatable and hence some portions of the kernel module's code space may be modified by the module loading function. Accordingly, the cryptographic hash of a loaded kernel module will be different depending on where it is relocated. To solve this problem, we perform an off-line, a priori profiling of the legitimate kernel module binaries. For each known good module we calculate the cryptographic hash by excluding the portions of the module that will be changed during relocation. In addition, we store a list of bytes affected by relocation so that the same procedure can be repeated by NICKLE during runtime hash evaluation of the same module.

Although the implementation of NICKLE requires certain guest OS-specific information, it does *not* require modifications to the guest OS itself. Still, for a closed-source guest OS (e.g., Windows), lack of information about kernel bootstrapping and dynamic kernel code loading may lead to certain limitations. For example, not knowing the timing and "signature" of dynamic (legal) kernel code loading events in Windows, the current implementation of NICKLE relies on the administrator to designate a time instance when all authorized Windows kernel code has been loaded into the standard memory. Not knowing the exact locations of the kernel code, NICKLE then copies the standard memory to the shadow memory, hence creating a "gold standard" against which to compare future kernel code execution. From this time on, NICKLE can transparently protect the Windows OS kernel from executing any unauthorized kernel code. Moreover, this limited implementation can be made complete when the relevant information becomes available through vendor disclosure or reverse engineering.

5.3.2 Flexible Response

In response to an attempt to execute an unauthorized instruction in the kernel space, NICKLE provides two response modes. Our initial implementation of NICKLE simply re-routes the instruction fetch to the shadow memory for a string of zeros (break mode). This causes a Linux guest OS to trigger a kernel fault and terminate the offending process. Windows reacts to the NICKLE response by immediately halting with a blue screen – a less graceful outcome.

In search of a more flexible response mode, we find that by rewriting the offending instructions at runtime (rewrite mode), NICKLE can respond in a less disruptive way. We also observe that most kernel rootkits analyzed behave the following way: They first insert malicious code into the kernel space; then they somehow ensure their code is `call`'d as a function. With this observation, we let NICKLE dynamically replace the code with `return -1;`, which in x86 assembly is: `mov $0xffffffff, %eax; ret`. The main kernel text or the kernel module loading process will interpret this as an error and gracefully handle it: Our experiments with Windows 2K/XP, Linux 2.4, and Linux 2.6 guest OSes all confirm that NICKLE's rewrite mode is able to handle the malicious kernel code execution attempt by triggering the OS to terminate the offending process without causing a fault in the OS.

5.3.3 Porting Experience

We have experienced no major difficulty in porting NICKLE to another VMM. The NICKLE implementations in both VMMs is lightweight: The SLOC (source lines of code) added to implement NICKLE in QEMU+KQEMU and VirtualBox are 853 and 762, respectively. As mentioned earlier, we first implemented NICKLE in QEMU+KQEMU.

The VirtualBox port is more complicated than the QEMU port. VirtualBox is a software VMM, but attempts to execute as much guest code (both user and kernel) as possible directly on the host processor in user mode. In the event that a piece of

guest kernel code cannot be executed directly, VirtualBox makes use of parts of the QEMU source code (namely the recompiler) to do binary translation. Our original port simply reused the NICKLE code for QEMU+KQEMU and modified VirtualBox to execute all kernel code using the QEMU recompiler. This caused performance degradation because of the speed difference between native and recompiler-based executions.

To achieve better performance, we used the following optimization: If a kernel page contains nothing but verified kernel code, then the code from the page will be executed directly on the host processor; For a kernel page mixed with both code and data, the execution will be passed off to the recompiler and the related memory requests will be mediated. This technique can result in significant performance gains: in the kernel compilation test (Section 5.4), NICKLE before optimization incurred a 50% slowdown while after optimization it is reduced to 7.06%. Rhe VirtualBox port is more difficult because of the complexity of the VMM itself. As such we still consider this port to be a proof of concept and reasonable indicator of performance, but additional time would be required to further reduce the performance overhead and make it comparable to the QEMU port.

5.4 NICKLE Evaluation

5.4.1 Effectiveness against Kernel Rootkits

We have evaluated the effectiveness of NICKLE with 22 real-world kernel rootkits. They consist of ten Linux 2.4 rootkits, seven Linux 2.6 rootkits, and five Windows rootkits[6] that can infect Windows 2000 and/or XP. The selected rootkits cover the main attack platforms and attack vectors thus providing a good representation of the state-of-the-art kernel rootkit technology. They were chosen because they were able to run successfully on our testing platform. Tables 5.1, 5.2, and 5.3 show our

[6]There is a Windows rootkit named hxdef or Hacker Defender, that is usually classified as a user-level rootkit. However, since hxdef contains a device driver which will be loaded into the kernel, we consider it a kernel rootkit in this dissertation.

Table 5.1
Effectiveness of NICKLE in detecting and preventing Linux 2.4 rootkits

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response			
			Rewrite Mode		Break Mode	
			Prevented?	Outcome	Prevented?	Outcome
Linux 2.4	adore 0.42, 0.53	LKM	✓	`insmod` fails	✓	Seg. fault
	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	knark	LKM	✓	`insmod` fails	✓	Seg. fault
	rkit 1.01	LKM	✓	`insmod` fails	✓	Seg. fault
	kbdv3	LKM	✓	`insmod` fails	✓	Seg. fault
	allroot	LKM	✓	`insmod` fails	✓	Seg. fault
	rial	LKM	✓	`insmod` fails	✓	Seg. fault
	Phantasmagoria	LKM	✓	`insmod` fails	✓	Seg. fault
	SucKIT 1.3b	`/dev/kmem`	✓	Installation fails silently	✓	Seg. fault

Table 5.2
Effectiveness of NICKLE in detecting and preventing Linux 2.6 rootkits

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response			
			Rewrite Mode		Break Mode	
			Prevented?	Outcome	Prevented?	Outcome
Linux 2.6	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	eNYeLKM v1.2	LKM	✓	`insmod` fails	✓	Seg. fault
	sk2rc2	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	superkit	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	mood-nt 2.3	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	override	LKM	✓	`insmod` fails	✓	Seg. fault
	Phalanx b6	`/dev/mem`	✓	Installation crashes	✓	Seg. fault

Table 5.3
Effectiveness of NICKLE in detecting and preventing Windows rootkits

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response			
			Rewrite Mode		Break Mode	
			Prevented?	Outcome	Prevented?	Outcome
Windows 2K/XP	FU	DKOM[a]	✓	Driver loading fails	✓	BSOD[b]
	FUTo	DKOM	✓	Driver loading fails	✓	BSOD
	he4hook 215b6	Driver	✓	Driver loading fails	✓	BSOD
	hxdef 1.0.0 revisited	Driver	partial[c]	Driver loading fails	✓	BSOD
	NT Rootkit	Driver	✓	Driver loading fails	✓	BSOD

[a] A common rootkit technique which directly manipulates kernel objects

[b] "Blue Screen Of Death"

[c] The in-kernel component of the Hacker Defender rootkit fails

experimental results: NICKLE is able to detect and prevent the execution of malicious kernel code in *all* experiments using both rewrite and break response modes. In the following, we present details of two representative experiments, SucKIT and FU.

The SucKIT rootkit [101] for Linux 2.4 infects the Linux kernel by directly modifying the kernel through the `/dev/kmem` interface. During installation SucKIT first allocates memory within the kernel, injects its code into the allocated memory, and then causes the code to run as a function. Figure 5.3 shows NICKLE preventing the SucKIT installation. The window on the left shows the VM running RedHat 8.0 (with 2.4.18 kernel), while the window on the right shows the NICKLE output. Inside the VM, one can see that the SucKIT installation program fails and returns an error message "*Unable to handle kernel NULL pointer dereference*". This occurs because NICKLE (operating in break mode) foils the execution of injected kernel code by fetching a string of zeros from the shadow memory, which causes the kernel to terminate the rootkit installation program. Interestingly, when NICKLE operates in rewrite mode, it rewrites the malicious code and forces it to return -1. However, it seems that SucKIT does not bother to check the return value and so the rootkit installation fails silently and the kernel-level functionality does not work.

In the right-side window in Figure 5.3, NICKLE reports the authentication and shadowing of sequences of kernel instructions starting from the initial BIOS bootstrap code to the kernel text as well as its initialization code and finally to various legitimate kernel modules. In this experiment, there are five legitimate kernel modules, *parport.o*, *parport_pc.o*, *ieee1394.o*, *ohci1394*, and *autofs.o*, all authenticated and shadowed. The code portion of the kernel module begins with an offset of `0x60` bytes in the first page. The first `0x60` bytes are for the kernel module header, which stores pointers to information such as the module's name, size, and other entries linking to the global linked list of loaded kernel modules. This is another example of *mixed kernel pages* with code and data in Linux (Section 5.2.2).

The FU rootkit [102] is a Windows rootkit that loads a kernel driver and proceeds to manipulate kernel data objects. The manipulation will allow the attacker to hide

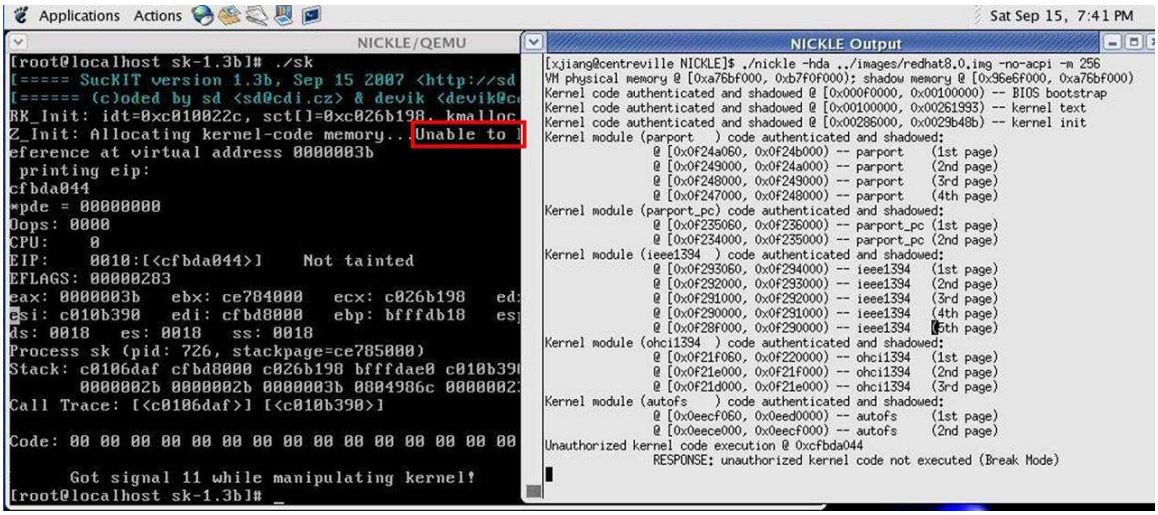


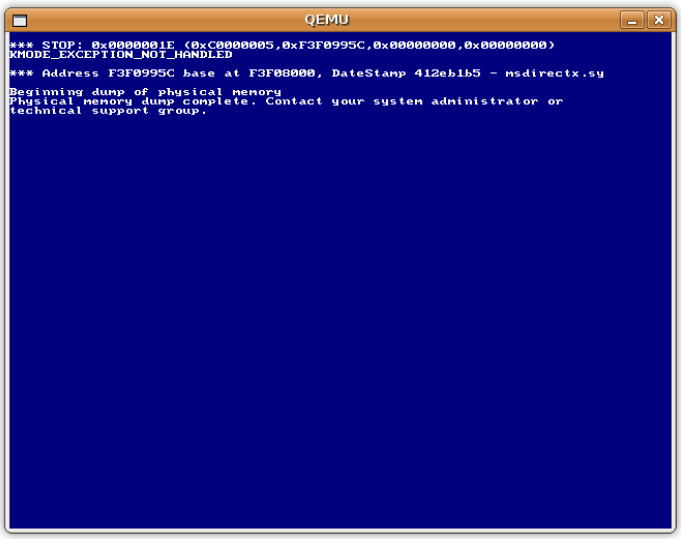
Figure 5.3. NICKLE/QEMU+KQEMU foils the SucKIT rootkit (guest OS: RedHat 8.0)

certain running processes or device drivers loaded in the kernel. When running FU on NICKLE, the driver is unable to load successfully as the driver-specific initialization code is considered unauthorized kernel code. Figure 5.4 compares NICKLE's two response modes against FU's attempt to load its driver. Under break mode, the OS simply breaks with a blue screen. Under rewrite mode, the FU installation program fails ("Failed to initialize driver.") but the OS does not crash.

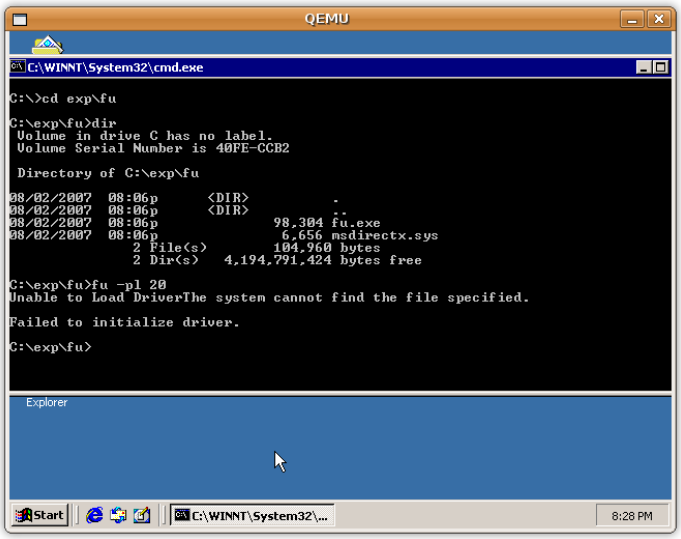
5.4.2 Impact on Performance

To evaluate NICKLE's impact on system performance we have performed benchmark based measurements on both VMMs – with and without NICKLE. The physical host in our experiments has an Intel 2.40GHz processor and 3GB of RAM running Ubuntu Linux 7.10. QEMU version 0.9.0 with KQEMU 1.3.0pre11 or VirtualBox 1.5.0 OSE is used where appropriate. The VM's guest OS is Redhat 8.0 with a custom compile of a vanilla Linux 2.4.18 kernel and is started in uniprocessor mode with the default amount of memory (256MB for VirtualBox and 128MB for QEMU+KQEMU). Table 5.4 shows the software configuration for the measurement. For the Apache benchmark, a separate machine connected to the host via a dedicated gigabit switch is used to launch ApacheBench. When applicable, benchmarks are run 10 times and the results are averaged.

Three application-level benchmarks (Table 5.5) and one micro-benchmark (Table 5.6) are used to evaluate the system. The first application benchmark is a kernel compilation test: A copy of the Linux 2.4.18 kernel is uncompressed, configured, and compiled. The total time for these operations is recorded and a lower number is better. This test is commonly used to help evaluate system performance. Second, the `insmod` benchmark measures the amount of time taken to insert a module (in this case, the *ieee1394* module) into the kernel and again lower is better. This benchmark is included to illustrate the amount of time it takes to authenticate and copy a module, something we believe will show the system's worst performance. Third,



(a) Under break mode



(b) Under rewrite mode

Figure 5.4. Comparison of NICKLE/QEMU+KQEMU's response modes against the FU rootkit (guest OS: Windows 2K)

Table 5.4
Software configuration for performance evaluation

Item	**Version**	**Configuration**
Redhat	8.0	Using Linux 2.4.18
Kernel	2.4.18	Standard kernel compilation
ApacheBench	2.0.40-dev	`-c3 -t 60 <url/file>`
Unixbench	4.1.0	`-10 index`
Apache	2.0.59	Using the default high-performance configuration file

the ApacheBench program is used to measure the VM's throughput when serving requests for a 16KB file. In this case, higher is better. Apache provides a strongly I/O bound workload and is thus included. Finally, the Unixbench micro-benchmark is executed to evaluate the more fine-grained performance impact of NICKLE. Unixbench provides a variety of tests useful in determining causes of slowdown. The numbers reported in Table 5.6 are an index where higher is better. It should be noted that the benchmarks are meant primarily to compare a NICKLE-enhanced VMM with the corresponding unmodified VMM. These numbers are not meant to compare different VMMs (such as QEMU+KQEMU vs. VirtualBox).

The QEMU+KQEMU implementation of NICKLE exhibits low overhead in most tests. A few of the benchmark tests show a slight performance gain for the NICKLE implementation, but we consider these results to signify that there is no noticeable slowdown caused by NICKLE for that test. From Table 5.5 it can be seen that both the kernel compilation and Apache tests come in below 1% overheard. The `insmod` test has a modest overhead, 7.3%, primarily because NICKLE must calculate and verify the hash of the module prior to copying it into the shadow memory. Given how infrequently kernel module insertion occurs in a running system, this overhead is not a concern. The Unixbench tests in Table 5.6 further testify to the efficiency

Table 5.5
Application benchmark results

	QEMU+KQEMU			VirtualBox		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Kernel Compiling	231.490s	233.529s	0.87%	156.482s	168.377s	7.06%
`insmod`	0.088s	0.095s	7.34%	0.035s	0.050s	30.00%
Apache	351.714 req/s	349.417 req/s	0.65%	463.140 req/s	375.024 req/s	19.03%

Table 5.6
Unixbench results (for the first two data columns, higher is better)

	QEMU+KQEMU			**VirtualBox**		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Dhrystone	659.3	660.0	-0.11%	1843.1	1768.6	4.04%
Whetstone	256.0	256.0	0.00%	605.8	543.0	10.37%
Execl	126.0	127.3	-1.03%	205.4	178.2	13.24%
File copy 256B	45.5	46	-1.10%	2511.8	2415.7	3.83%
File copy 1kB	67.6	68.2	-0.89%	4837.5	4646.9	3.94%
File copy 4kB	128.4	127.4	0.78%	7249.9	7134.3	1.59%
Pipe throughput	41.7	40.7	2.40%	4646.9	4590.9	1.21%
Process creation	124.7	118.2	5.21%	92.1	85.3	7.38%
Shell scripts (8)	198.3	196.7	0.81%	259.2	239.8	7.48%
System call	20.9	20.1	3.83%	2193.3	2179.9	0.61%
Overall	106.1	105.0	**1.01%**	1172.6	1108.7	**5.45%**

of the NICKLE implementation in QEMU+KQEMU, with the worst-case overhead of any test being 5.21% and the overall overhead being 1.01%. The low overhead of NICKLE is because NICKLE's modifications to the QEMU control flow only take effect while executing kernel code (user-level code is executed by the unmodified KQEMU accelerator).

The VirtualBox implementation has a more noticeable overhead than the QEMU implementation, but still runs below 10% for the majority of the tests. The kernel compilation test, for example, exhibits about 7% overhead; while the Unixbench suite shows a little less than 6% overall. The Apache test shows the highest overheard, a 19.03% slowdown. This can be attributed to the heavy number of user/kernel mode switches that occur while serving web requests. It is during these mode switches that the VirtualBox implementation does its work to ensure only verified code will be executed directly, hence incurring overhead. The `insmod` test shows a large performance degradation, coming in at 30.0%. This is because module insertion on the VirtualBox implementation entails the VMM leaving native code execution as well as verifying the module. However, this is not a concern as module insertion is an uncommon event at runtime. Table 5.6 shows that the worst performing Unixbench test (Execl) results in an overhead of 13.24%. This result is most likely because of a larger number of user/kernel mode switches that occur during that test.

In summary, our benchmark experiments show that NICKLE incurs minimal to moderate impact on system performance, relative to that of the respective original VMMs.

5.5 Discussion

In this section, we discuss several issues related to NICKLE. First, the goal of NICKLE is to prevent unauthorized code from executing in the kernel space, but not to protect the integrity of kernel-level control flows. This means that it is possible for an attacker to launch a "return-into-libc" style attack within the kernel by lever-

aging only the existing authenticated kernel code. As mentioned previously, work by Shacham [16] and Buchanan et al. [17] models a powerful attacker who can execute virtually arbitrary code using only a carefully crafted stack that causes jumps and calls into existing code. Fortunately, this approach cannot produce *persistent* code to be called on demand from other portions of the kernel. And Petroni et al. [47] found that 96% of the rootkits they surveyed require persistent code changes. From another perspective, an attacker may also be able to directly or indirectly influence the kernel-level control flow by manipulating certain non-control data [18]. However, without its own kernel code, this type of attack tends to have limited functionality. For example, all four stealth rootkit attacks described in [103] need to execute their own code in kernel space and hence will be defeated by NICKLE. Meanwhile, solutions exist for protecting control flow integrity [43, 47, 104] and data flow integrity [105], which can be leveraged and extended to complement NICKLE.

Second, the current NICKLE implementation does not support self-modifying kernel code. This limitation can be removed by intercepting the self-modifying behavior (e.g., based on the translation cache invalidation resulting from the self-modification) and re-authenticating and shadowing the kernel code after the modification.

Third, NICKLE currently does not support kernel page swapping. Linux does not swap out kernel pages, but Windows does have this capability. To support kernel page swapping in NICKLE, it would require implementing the introspection of swap-out and swap-in events and ensuring that the page being swapped in has the same hash as when it was swapped out. Otherwise an attacker could modify swapped out code pages without NICKLE noticing. This limitation has not yet created any problem in our experiments, where we did not encounter any kernel level page swapping.

Fourth, targeting kernel-level rootkits, NICKLE is ineffective against user-level rootkits. However, NICKLE significantly elevates the trustworthiness of the guest OS, on top of which anti-malware systems can be deployed to defend against user-level rootkits more effectively.

Fifth, the deployment of NICKLE increases the memory footprint for the protected VM. In the worst case, memory shadowing will double the physical memory usage. As future work, we can explore the use of demand-paging to effectively reduce the extra memory requirement to the actual amount of memory needed. Overall, it is reasonable and practical to trade memory space for elevated OS kernel security.

Finally, we point out that NICKLE assumes a trusted VMM to achieve the "NICKLE" property. This assumption is needed because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. We also acknowledge that a VM environment can potentially be fingerprinted and detected [106, 107] by attackers so that their malware can exhibit different behavior [108]. We could improve the fidelity of the VM environment (e.g., [109, 110]) to thwart some of the VM detection methods, however some researchers report this to be infeasible in the general sense [111]. Meanwhile, as virtualization continues to gain popularity, the concern over VM detection may become less significant as attackers' incentive and motivation to target VMs increases.

5.6 Hardware Support

While this chapter discusses a software VMM-based approach for constructing an SMA at the kernel level, simple modifications to the processor architecture could significantly lessen the performance overhead.

Some processors contain hardware support for virtualization designed to greatly increase the efficiency of x86 virtualization. One of these features, Nested Paging, provides an additional level of paging at the virtualization level to translate guest-physical addresses to actual physical addresses. A nested page table (NPT) provides the mappings. For a detailed description, see [112].

A processor with Nested Paging support can be modified to better accommodate building an SMA by adding the nested CR3 registers needed to accommodate two nested page tables: One for kernel code and one for everything else. With this sort

of hardware support the virtual machine monitor would setup both sets of NPTs for a running operating system to execute it on an SMA.

This sort of support would significantly reduce or remove the overhead required to mediate operating system memory accesses, however overhead related to kernel code authentication and copying would be unchanged. In addition, the system would still need to make use of a virtual machine monitor to handle the authentication and copying. (The VMM, however, could be significantly smaller.)

This particular hardware design is discussed because it is a straightforward extension of the software design presented.

5.7 Summary

In this chapter we have demonstrated the efficacy of the SMA for the prevention of kernel level rootkits. We have presented the design, implementation, and evaluation of NICKLE, a VMM-based SMA approach that transparently detects and prevents the launching of kernel rootkit attacks against guest VMs. NICKLE achieves the "NICKLE" guarantee, which foils the common need of existing kernel rootkits to execute their own unauthorized code in the kernel space. NICKLE makes use of a VMM-based SMA and achieves guest transparency through the guest memory access indirection technique. NICKLE's portability has been demonstrated by its implementation in two VMM platforms. Our experiments show that NICKLE is effective in preventing 22 representative real-world kernel rootkits that target a variety of commodity OSes. Our measurement results show that NICKLE adds less than 8% overheard to the QEMU platform.

6 SMA-ASSISTED PROFILING OF INJECTED CODE

We have discussed the use of an SMA to prevent code injection at both the user-level and the kernel-level, we now turn to demonstrating the applicability of an SMA for profiling the behavior of injected code.

We have demonstrated this concept to a small extent at the user-level in Chapter 4 with the observe and forensic response modes. The SMA is particularly suitable as the basis for a profiler because it allows for efficient and effective comparisons between the code and data memory spaces.

In this chapter we will discuss a technique for using NICKLE's VMM-based SMA as a foundation for constructing a kernel rootkit profiler [113].

6.1 Introduction

Despite recent research efforts in kernel rootkit detection [44,46,47,50] and kernel rootkit prevention [51, 91], less attention has been given to *kernel rootkit profiling*– the revelation of key aspects of a kernel rootkit's behavior. It is desirable that such profiles be generated on-the-fly in "live" systems such as honeypots. Kernel rootkit profiles are valuable in the design of effective solutions to kernel rootkit detection, damage mitigation, and kernel integrity protection. We define a kernel rootkit profile as being comprised of the following four aspects:

- *Hooking behavior*: the way the kernel rootkit hijacks the kernel's control flow, if any, during the rootkit's installation. Typically, such hijacking is done by modifying hooks (e.g., function pointers) in the kernel. Note that it is not uncommon for rootkits to install hooks within various kernel objects, including kernel code or dynamically allocated kernel objects [114].

- *Targeted kernel objects*: the kernel objects accessed by the rootkit, such as those read or modified by the rootkit. Similar to hooking behavior, a targeted kernel object may be dynamic. A classic example is the task list, maintained by the OS kernel for accounting purposes but often manipulated by rootkits for hiding purposes.
- *User-level impacts*: the affected user-level applications whose execution may be directly affected by the execution of rootkit code. We do not derive a complete list of affected applications. Instead, we focus on a corpus of commonly-used system utilities (e.g., `ps`, `ls`, `netstat`, etc.) that retrieve important system information and are therefore often targeted by kernel rootkits.
- *Injected code*: the kernel rootkit code injected into the kernel memory address space for execution. The injected code should be accurately located at runtime and extracted for later forensic analysis.

A number of recent have been reported towards kernel rootkit profiling [58,60–62]. Despite their usefulness, the current approaches leave more to be desired in their capabilities: (1) Some approaches require prior availability of the kernel rootkit code and knowledge that the rootkit attack is going to occur. However, such requirements make it difficult to profile zero-day kernel rootkits. (2) The current profiling techniques only focus on one aspect of rootkit behavior (e.g., hooking behavior) or on one stage of a rootkit's life cycle (e.g., installation or execution but not both). (3) The key techniques used in the existing approaches such as system-wide tainting or slicing have well-known limitations and challenges that need to be overcome. For example, taint-based information flow tracking can be circumvented by various control-flow evasion schemes [59].

To overcome the above limitations we present PoKeR (*P*rofiler *o*f *Ke*rnel *R*ootkits), a virtualization-based kernel rootkit profiler that generates multi-aspect kernel rootkit profiles during rootkit execution. PoKeR is designed to be deployed in a system that can tolerate high performance overhead, such as a honeypot which is subject to rootkit attacks in the wild. A PoKeR-enabled system executes normally until a kernel rootkit

is installed and ready to execute malicious code injected into the kernel. At that point, PoKeR switches the system (a virtual machine or VM) to a rootkit profiling mode and applies a strategy called "*combat tracking*"[1] to automatically track and determine the kernel objects, static and dynamic, that are being targeted by the kernel rootkit. In addition, when injected rootkit code is being executed, PoKeR records the relevant system call contexts and infers potential effects on user-level applications.

We have developed a prototype of PoKeR and used it to profile 10 representative real-world kernel rootkits that exhibit a broad range of attack methodologies. This includes basic system call table hooking, the more advanced technique of direct kernel object manipulation [115], manipulation of function pointers inside dynamic kernel data objects [114], and others. The profiles generated by PoKeR capture multiple aspects of the rootkit's behavior and permit unique insights into each rootkit's characteristics. We have also measured the performance of our QEMU-based prototype and found that it degrades virtualization system performance between 3x and 6x during profiling, with the virtualization system itself adding an additional slowdown of 3.8x above and beyond that of the physical host.

The contributions of this chapter are as follows:

- We identify four key aspects of kernel rootkit behavior and use them to characterize and profile existing kernel rootkits.
- We define the concept of an *instantaneous rootkit detection system* and discuss how a VMM-based SMA can be used as one to generate a *detection point* to trigger rootkit profiling.
- We propose a technique called *combat tracking* to determine the identity and type information of rootkit-targeted kernel objects, even if they are dynamically allocated from the kernel heap.
- We develop a PoKeR prototype and present the evaluation results with 10 representative real-world rootkits. The obtained rootkit profiles provide useful

[1]Combat tracking, in war, is the art of hunting the enemy by following the signs he leaves behind as he moves. In PoKeR, we intend to follow the trail the rootkit leaves behind.

insights into rootkit behavior, some of which are difficult to obtain without PoKeR, despite in-depth analysis.

6.2 Assumptions

Similar to Chapter 5, in this chapter we assume that a kernel rootkit has the same memory access privileges as the OS kernel itself. If the OS can read from or write to a memory location, so can the rootkit. This also means that the rootkit does not have privileges higher than that of the OS, such as those of a virtual machine monitor (VMM). The rootkit is free to modify any kernel objects, whether static or dynamic.

We also assume that the rootkit requires the execution of *injected code* at the kernel's privilege level. We do not, however, require that the injected code be persistent throughout the life cycle of the rootkit attack. We refer to a kernel rootkit that requires the execution of injected code at the kernel's privilege level as a *code injection kernel rootkit*. For ease of presentation, we will use the term kernel rootkit to refer to a code injection kernel rootkit. This assumption is realistic. Petroni et al. [47] surveyed 25 kernel rootkits and none of them violate our assumptions. In particular, all 25 rootkits make use of injected code in the kernel space, and 24 of them require injected code to be persistent throughout their lifetime.

With regards to PoKeR itself, we assume that it has access to the OS kernel source code for static analysis, or to debugging symbols and type information for an already compiled kernel binary. We also assume that the system PoKeR is running on can tolerate high performance overhead during profiling.

6.3 Design

Figure 6.1 shows the overall architecture of PoKeR. As highlighted in the figure, PoKeR has two main components:

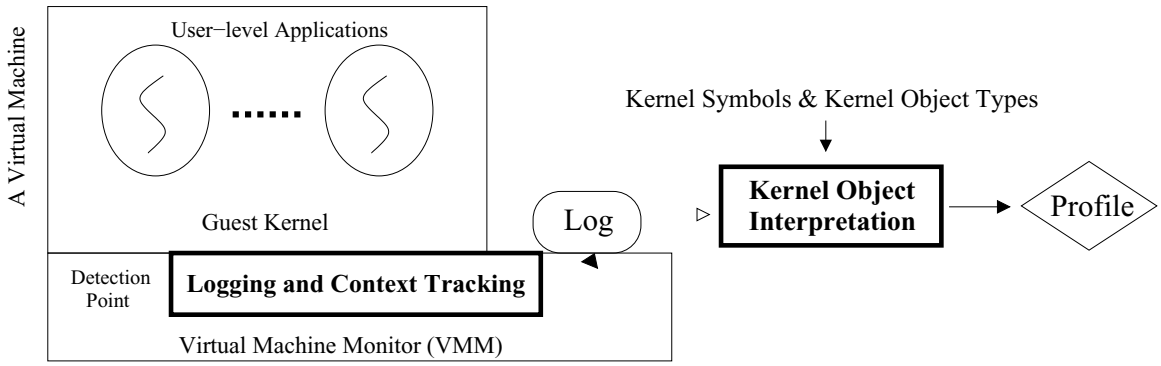


Figure 6.1. VMM-based PoKeR architecture

- The *Logging and Context Tracking* module resides inside the VMM and, once activated, collects runtime execution traces of malicious rootkit code. The execution trace is saved outside the target VM and contains information such as rootkit instructions executed, corresponding memory reads and writes, and associated execution context. The logging of execution context will be helpful later in assessing the user-level impacts of the rootkit attack. Note that the activation of this module requires a detection point, which we will discuss in Section 6.3.1.

- The *Kernel Object Interpretation* module processes the collected execution trace and resolves read and write target addresses into the kernel objects read or manipulated by a rootkit. The dynamic nature of certain kernel objects significantly complicates the interpretation procedure.

There are three key challenges and techniques associated with the design of PoKeR. They will be presented in the following three subsections.

6.3.1 Switching to Profiling Mode

As mentioned in Section 6.1, PoKeR is primarily designed to be used in environments that can tolerate high overhead. A PoKeR-enabled system has two modes of

operation. The first mode, *detection mode*, is its initial state. While in this mode, an instantaneous rootkit detection system (defined below) watches for kernel rootkit execution. Most of PoKeR's rootkit profiling features are disabled during detection mode. The other mode, *profiling mode*, starts right at the *detection point*, when the instantaneous rootkit detection system reports that a kernel rootkit attack is about to occur. In profiling mode, PoKeR enables its profiling features and logs the rootkit's actions at a fine granularity, such as instruction execution, system calls, and memory reads and writes. PoKeR will then generate the rootkit's profile according to the four aspects defined in Section 6.1.

To ensure that all of a rootkit's actions are profiled properly, the detection point must be generated before the first rootkit instruction is about to execute in the kernel. We refer to a detection system capable of meeting this strict time constraint as an *instantaneous detection system*. We leverage NICKLE's VMM-based SMA to serve as the instantaneous detection system that generates kernel rootkit detection points for PoKeR.

Turning the original NICKLE into an instantaneous rootkit detection system for PoKeR is straightforward. Instead of simply blocking rootkit code execution, the system will allow the code to be executed unhindered from the standard memory. This is similar to observe mode in the user-level code injection prevention system as described in Chapter 4. During a guest kernel instruction fetch the contents of the standard and shadow memory are compared to determine if the same instruction exists in both. If a kernel instruction that is about to be fetched exists in the standard memory but not the shadow memory (or if the contents simply differ) then unauthorized code is about to be executed at the kernel level. This serves as PoKeR's detection point, and the system can be switched to profiling mode.

Given that we know an instruction is malicious prior to executing it and the SMA has a copy of both the original and modified memory, we have the unique opportunity to identify and extract the malicious rootkit code. It can then be analyzed further later on, such as by static analysis. To aid in this, we also record the order in

which the instructions were executed. In addition, the malicious code identification capability may allow profiling mode to turn on and off during profiling – on when rootkit instructions are executed and off when authenticated kernel instructions are executed. The dynamic toggling between detection mode (faster) and profiling mode (slower) may result in better rootkit profiling efficiency.

6.3.2 Tracking Targeted Kernel Objects

Once kernel rootkit execution is detected and the profiling mode of PoKeR is switched on, it is necessary to keep track of all kernel objects manipulated by the kernel rootkit. The rootkit may, for example, traverse the entire process list looking for an entry with a specific PID to remove. Or, it may change key values in a TCP data structure within the kernel to mask the sending of data to a remote location. It is important that PoKeR be able to determine, upon the execution of a rootkit instruction, which kernel object is being read or modified. This is challenging because PoKeR operates at the VMM level, which does not directly provide a semantic view of the guest kernel objects. Unfortunately, current virtual machine introspection techniques [49, 50, 116] do not support such a "reverse lookup" (namely, given a memory address, identify the corresponding kernel object).

A list of the rootkit's reads and writes is simple to obtain using PoKeR's logging and context tracking module, as it simply logs all reads and writes performed by the rootkit code. However, determining which kernel objects a rootkit is modifying is complicated because a large number of kernel objects are dynamically allocated. For example, we may know that a rootkit is modifying memory at address `0xc6600856`, but if that address is located within the kernel's heap there is no simple way to determine what object it is. (This is one reason that a simple symbolic debugger cannot be used to track kernel objects.) This is in contrast to statically allocated kernel objects, whose addresses can be easily determined at compile time. To handle dynamically allocated kernel objects, we need to create an *address-to-dynamic object*

map that can be used to translate memory addresses into the kernel objects they are a part of.

One key observation that helps in creating this address-to-dynamic object map is that all dynamically allocated kernel objects must be accessible in some way from global variables or CPU registers. If one imagines kernel objects as a graph where the edges are pointers, then all objects will be transitively reachable from at least one global variable. If an object is not reachable in this way, then the kernel itself will not be able to access it and the object cannot be used. A similar observation has also been made in previous work on both garbage collection [117] and state-based control-flow integrity [47]. A brute force approach for mapping an address to a dynamic object would be to search the entire memory graph. This would be inefficient.

To support the address-to-dynamic object mapping in a more efficient way, we propose a technique called "combat tracking." The key observation in our combat tracking technique is that for a kernel rootkit to find the address of a dynamically allocated kernel object, it will first traverse to it from a statically allocated one. The rootkit, much like PoKeR, is naturally ignorant of the layout of dynamic kernel objects, and therefore will do a series of reads of kernel memory to reach the objects. By tracking a rootkit through its series of reads, we can dynamically build up an address-to-dynamic object map for PoKeR to look up a corresponding dynamic kernel object when given a memory address.

Algorithm 6.2 shows the combat tracking algorithm executed by PoKeR's kernel object interpretation module. The algorithm assumes the availability of an initial map of static objects and uses that, combined with the rootkit's reads, to build the map of dynamic objects on the fly. (In our prototype, the static kernel object map as well as the object type definitions come from a copy of the kernel compiled with debug symbols.) The first step in the algorithm is to determine the type of data at the address being read. We first query the static object map to see if the object is a global object and if that fails then we check our dynamic object map to see if we have previously added this address to the map. Once we find the type of the object

```
Input: Address of read (addr), Value read (val)
1  if addr in static map then
2      // Query the static data for type information of the address;
3      type ← static_objects(addr);
4  else if addr in dynamic map then
5      // Query the dynamic map instead;
6      type ← query_dynamic_map(addr);
7  else
8      // No type information known;
9      return;
10 end
11 if type is pointer then
12     // If we have a pointer, val is the address of a kernel object;
13     d_typ ← dereference(type);
14     add_dynamic_map(val, d_type);
15 end
```

Figure 6.2. Combat tracking algorithm

being read, we determine if it is a pointer. We care about pointers because if a read occurs on a pointer object, then the value of the read corresponds to the address of a kernel object. This may be a kernel object we have not seen before, and it can be used to further build the dynamic map. Given this, in the event the rootkit did read a pointer, we determine the value read by the rootkit (the address of the new object) as well as the de-referenced type of the pointer (the type of the new object) and we add this information to the dynamic map. In this way we progressively build up the address-to-dynamic object map based on the rootkit's reads.

To illustrate combat tracking, let us consider an example. Figure 6.3 is a simplified representation of the process list maintained in the Linux kernel. There is one global

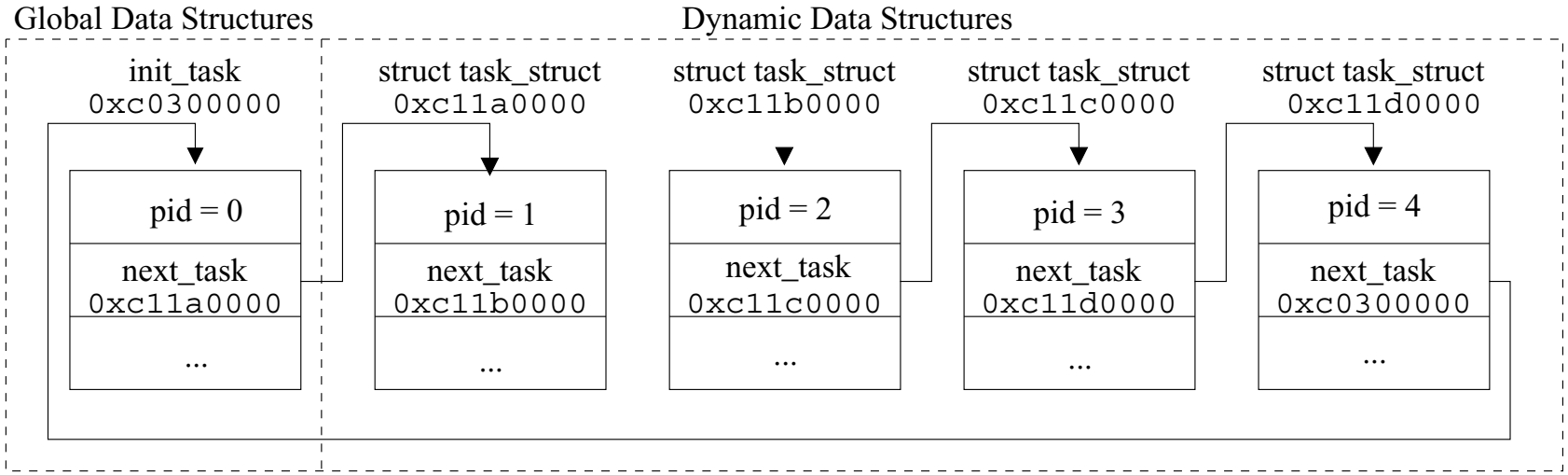


Figure 6.3. A simplified example of a Linux process list

data structure at address `0xc0300000`, `init_task`, which forms the head of a linked list of dynamically allocated `struct task_struct` structures. If a rootkit were to try to find the `task_struct` for pid 3, it would do the following. First, it would read address `0xc0300004` to find the `next_task` pointer in the global `task_struct`. It would receive `0xc11a0000`, the address of the next structure. Next, it would read the pid of that next structure at address `0xc11a0000`, and when it found that it was not 3, it would read `0xc11a0004` to find the next `task_struct` to search. It would repeat this procedure until it found pid 3 in the `task_struct` at address `0xc11c0000`. From there it may modify a variable in that data structure (e.g. at address `0xc11c0008`) to perform some sort of kernel object manipulation.

Without combat tracking, we would only know that the rootkit did a write at address `0xc11c0008` and we would have no way of knowing what kind of data was at the address. With combat tracking, given the entire chain of reads, the dynamic map would be built: When the rootkit first reads the `next_task` element of `init_task`, a query of the initial static map tells us that the read corresponds to an object of type `struct task_struct *`. Given this knowledge, combined with the knowledge that the rootkit reads `0xc11a0000` from that location, we know that address `0xc11a0000` contains a `struct task_struct` and add it to our dynamic map. When the rootkit later reads the `next_task` pointer from that dynamic data structure, we know (from the previous read) that the read is for another pointer of type `struct task_struct *` and can add that element of the linked list to our dynamic map as well. We continue on in this fashion until we have a map of all the data structures the rootkit has read. Later, when the write to address `0xc11c0008` occurs, we can check the dynamic map to know that the address is part of a `task_struct` and determine which element of the data structure is being modified.

We do not keep track of a kernel object's lifetime and remove its entry from the dynamic map after its de-allocation. The entry will still exist in the map despite there being no object at that location. Such a "stale entry" does not matter, however, because the rootkit should not access a deallocated kernel object. (If it does, it is

most likely a programming error.) If a new object is ever allocated at a previously used address, then the chain of rootkit reads to the new object will result in the stale entry being replaced by a new entry that corresponds to the new object.

6.3.3 Discovering Rootkit Hooking and User-Level Impacts

For many kernel rootkits, one key reason for manipulating a specific subset of kernel objects is to eventually hijack the kernel's control flow so that the rootkit can somehow affect the execution state of the running kernel. The hijacking behavior is typically accomplished by modifying function pointers, many of which may be stored in dynamically allocated objects within the kernel's heap. To reveal a rootkit's hooking behavior, it is vital that we be able to find these hooks as they are being installed. It is also possible for the rootkit to directly modify legitimate code to force a call to the rootkit code. Fortunately, both types of changes can be thought of as a subset of the kernel object tracking problem. Tracking modifications to existing code is similar to tracking modifications to static objects; whereas tracking function pointer modifications is simply a part of tracking object modifications using combat tracking – the main reason being that the modified function pointers belong to certain kernel objects.

As an example, consider a Linux kernel module (LKM) based rootkit with the goal of ensuring that files that end in the extension "`.hacker`" are never visible to a user. The attacker installs this malicious rootkit as a kernel module using the `insmod` command. The system copies the malicious module into memory and then runs the module's `init()` function. Before the first instruction from `init()` is executed, the instantaneous rootkit detection system generates a detection point which turns on PoKeR's profiling mode. Next, the rootkit's initialization function modifies the system call table so that the system call originally used to retrieve a directory listing is changed to point to a malicious function that ensures files ending in `.hacker` do not appear in the listing. The write to the system call table is logged and interpreted.

Thus the code's hooking point is discovered and the control flow modification made by the rootkit is profiled.

In addition to determining which function pointers get hijacked by a kernel rootkit, it is also desirable to determine how the modified kernel control flow will impact system calls made by user-level programs. This may help ascertaining which user-level programs are being targeted by a specific rootkit as well as giving us a general understanding of what the rootkit is trying to hide. For kernel rootkits that modify the system call table, such impact is obvious: explicitly modified table entries will result in hijacked control flow when the corresponding system calls are made. For rootkits that do not directly modify system call table entries, however, determining which system calls will be affected is less obvious.

To determine which system calls get their control flow hijacked at runtime, we need to be able to correlate the execution of malicious rootkit code with the execution of the system call that led to it. To accomplish this, PoKeR will track the execution of system calls and apply a virtual machine introspection technique [49] to determine the current process context, namely which process is making the system call. Note that by logging the starting point when a system call is made and the ending point when the system call returns, PoKeR can effectively keep track of the lifetime of the system call. If malicious code execution is detected, PoKeR will infer the current process context of the malicious code execution and determine if any ongoing system call has the same process context. If so, the control flow of that system call is hijacked.

6.4 Implementation

To validate our design, we have developed a prototype of PoKeR. In this section we will describe its implementation.

```
1: M - 0xc883d000
2: R - 0xc1548054 - 0xc154a000
3: C - 0xc6706000
4: W - 0xc6707f24 - 0xc6707f3c
5: SC - 619 - insmod - sys_write
6: E - 0xc883ea28 - 619
7: SR - 619 - insmod
```

Figure 6.4. Sample log entries generated by PoKeR

6.4.1 Instantaneous Rootkit Detection

As mentioned in Section 6.3.1, NICKLE provides our VMM-based SMA for instantaneous rootkit detection. We made use the QEMU port of NICKLE as a basis for PoKeR.

6.4.2 Logging and Context Tracking

Once NICKLE signals the detection of malicious kernel rootkit code, PoKeR enters profiling mode. In profiling mode all kernel instructions are interpreted using the built-in dynamic re-compiler (a virtualization technique based on efficient, dynamic translation of guest code into host code) in QEMU so that the rootkit's actions can be logged at a fine granularity.

A sample of the log is shown in Figure 6.4. It shows the seven different types of log entries. The `R` and `W` log lines (lines 2 and 4) signify that the malicious rootkit code is now reading or writing. The reads and writes are caught by extending the QEMU-translated VM memory access instruction to include a check on whether the instruction issuing the access is malicious. The first argument on the line is the memory address being read or written and the second argument is the corresponding memory content. The `E` log line (line 6) signifies the execution of rootkit code and is

generated by PoKeR while a malicious instruction is being translated for execution. The arguments are the address of the malicious instruction and the pid of the process context it is running in, respectively. The `M` log line (line 1) is emitted whenever a kernel module is loaded (as seen by virtual machine introspection that is a part of NICKLE) and signifies the base address of that module's kernel data structure. (The `M` log line is the one item logged before a detection point is raised.) The `C` log line (line 3) is used to signify the address of the task structure of the currently running process (`current` in Linux) and is output preceding a read or write from that task structure.[2] The `SC` and `SR` log lines (lines 5 and 7) signify the start and end, respectively, of a system call. The `SC` log line includes information about the pid, program name, and system call made and is generated by extending the binary translation of the system call interrupt (`int 0x80` and `sysenter`). The `SR` log line only conveys the pid and program name, and is generated during the kernel-to-user mode switch.

The `SC`, `SR`, and `E` log entries allow us to determine which system calls have their control flow hijacked by a kernel rootkit. This is done by correlating the system call log entries with the rootkit code execution entries via the process context information. We parse through the log file and track currently running system calls (they begin with an `SC` and end with an `SR`) for running processes. In the event an `E` log line for a given process occurs while there is an open system call in that process, we know that the system call's control flow has been hijacked.

As mentioned earlier, the malicious rootkit instructions executed are logged along with the order in which they are executed. Later, a customized disassembler [118] is used to combine these two pieces of information and produce a copy of the rootkit's executed code annotated with its order of execution.

[2] `current` in Linux is not an actual variable, it is instead a macro that derives the address of the task structure for the currently running process based on the runtime kernel stack. The address of `current` cannot be determined by static analysis and this hint is needed by the object tracker later on. We output `current` during rootkit reads and writes that involve the task structure for the currently running process.

6.4.3 Kernel Object Interpretation

Once the log file of memory accesses is available, it is important to translate these accesses into names and types of the corresponding kernel objects. To track both static and dynamic kernel objects as described in Section 6.3.2, static analysis must be performed on the kernel itself. PoKeR can then use this information in conjunction with the rootkit's memory reads to instantiate our combat tracking technique.

The Linux kernel is a large, complicated code base that makes traditional static analysis difficult. However, by compiling a copy of the kernel with debug symbols (the `-g` flag to `gcc`) the GNU debugger (`gdb`) [119] can be used to extract the types, names, and locations of all static kernel objects. We modified `gdb` to facilitate easier access to this information and query for static kernel object information.

PoKeR's kernel object interpretation module is written in Python and implements combat tracking. It uses `gdb` for static type information and progressively builds its own internal map of dynamic kernel objects by processing rootkit reads using the algorithm in Section 6.3.2. The rootkit's kernel object manipulation profile can then be produced by querying the static and dynamic kernel object maps in interpreting the rootkit's memory writes. Our implementation also facilitates manual type annotation to accommodate *union* types. For the current prototype *union*s are handled by having a human user decide a priori which type should be used when that specific union is encountered. Another possibility would be to bifurcate union decisions by inserting all possibilities into the dynamic map. This could, however, result in an explosion of search space in the map. We look to emerging work in the area of automatic type determination [120] to eventually automate the handling of unions.

6.5 Evaluation

In this section we present the results of using PoKeR to profile 10 real-world kernel rootkits and give a brief evaluation of PoKeR's performance. The 10 rootkits were chosen because they were able to execute in our testing environment. In our

Table 6.1
Summary of kernel rootkit profiling results using PoKeR (Part 1)

Name	**Code**	**Kernel Objects Modified**		**User-Level Impacts**	**Attack Type**
		Kernel Object	**Note**		

SucKIT 1.3b	1687 instr	`sys_call_table[59]`
		`system_call` at offset 47
		`tracesys` at offset 27
		`current->addr_limit`
		`current->flags`
			Function Pointer
			Code
			Code
			Data Object
			Data Object
				2 - fork
				3 - read
				4 - write
				5 - open
				6 - close
				11 - execve
				85 - readlink
				195 - stat64
				196 - lstat64
				220 - getdents64
					code change, syscall hook

rial	475 instr	`sys_call_table[3,5,6,141,167]`	Function Pointers	3 - read 5 - open 6 - close 167 - query_mod	syscall hook
rkit 1.01	12 instr	`sys_call_table[23]`	Function Pointer		syscall hook

knark 0.59	490 instr	`sys_call_table[2,3,11,37,54]`
		`sys_call_table[79,120,141,220]`
		`current->flags`
			Function Pointers
			Function Pointers
			Data Object
				2 - fork
				3 - read
				11 - execve
				54 - ioctl
				220 - getdents64
					syscall hook

kbdv3	30 instr	`sys_call_table[30,199]`
		`current->uid`
		`current->euid`
		`current->gid`
		`current->egid`
			Function Pointers
			Data Object
			Data Object
			Data Object
			Data Object
				199 - getuid32	syscall hook, DKOM

Table 6.2
Summary of kernel rootkit profiling results using PoKeR (Part 2)

Name	Code	Kernel Objects Modified		User-Level Impacts	Attack Type
		Kernel Object	**Note**		
adore 0.42	770 instr	`sys_call_table[2,4,5,6,18,37,39,84,106]` `sys_call_table[107,120,141,195,196,220]`	Function Pointers Function Pointers	2 - fork 4 - write 5 - open 6 - close 195 - stat64 196 - lstat64 220 - getdents64	syscall hook

adore 0.53	733 instr	`sys_call_table[1,2,6,26,37,39,120,141,220]`
		`proc_net->subdir->next->(...)->next->get_info`
		`proc_root_inode_operations->lookup`
			Function Pointers
			Function Pointer
			Function Pointer
				1 - exit
				2 - fork
				3 - read
				5 - open
				6 - close
				85 - readlink
				195 - stat64
				220 - getdents64
					syscall hook, data hook

adore-ng 0.56	785 instr	`proc_net->subdir->next->(...)->next->get_info`
		`proc_root_inode_operations->lookup`
		`proc_root_operations->readdir`
		`ext3_dir_operations->readdir`
		`ext3_file_operations->write`
		`unix_dgram_ops->recvmsg`
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
				3 - read
				5 - open
				85 - readlink
				195 - stat64
				220 - getdents64
					data hook

linuxfu	117 instr	`init_task->next_task->(...)->prev_task->next_task` `init_task->next_task->(...)->next_task->prev_task`	Data Object Data Object		DKOM

hp 1.0.0	100 instr	`pidhash[600]`
		`pidhash[600]->pid`
		`pidhash[600]->prev_task->next_task`
		`pidhash[600]->next_task->prev_task`
		`pidhash[600]->p_osptr->p_ysptr`
		`pidhash[600]->p_ysptr->p_osptr`
			Data Object
			Data Object
			Data Object
			Data Object
			Data Object
			Data Object
					DKOM

experiments, the host machine is an Intel Core 2 - 2.4GHz desktop running Ubuntu 8.10. The VMM is a modified version of QEMU 0.9.0 running with KQEMU enabled[3]. Our guest OS is RedHat 8.0[4] running a recompiled version of its stock kernel, Linux 2.4.18-14. The recompilation is needed to produce a version with debug symbols (Section 6.4.3.)

Tables 6.1 and 6.2 show an abbreviated summary of the profiling results. For each kernel rootkit, its profile consists of the four aspects described in Section 6.1. The first aspect, hooking behavior, is revealed by the modified function pointers in certain kernel objects shown in the table. The second aspect of the profile, targeted kernel objects, indicates which objects are of interest to a rootkit. Kernel objects read but not modified are part of this aspect of the profile, but are not shown in the table because of the sheer quantity of them.

The third aspect of the profile is the potential impact on user-level programs. Given that most rootkits have a primary goal of altering a system administrator's view of the OS, we ran a corpus of 10 system utility programs that retrieve system information that kernel rootkits tend to hide. Four of them, `w`, `who`, `uptime`, and `finger` are capable of showing information related to currently logged-in users. Two, `netstat` and `ifconfig`, reveal information about network usage. Another pair, `ls` and `bash`, can reveal the existence of files. Information about running processes can be obtained by `ps`. Finally, `lsmod` shows the list of installed kernel modules.

These 10 programs were run and tested to see how many of the system calls they made resulted in the execution of rootkit code. They do not, however, represent the execution of all possible system calls. While a program could be written to exercise all system calls, the enormous variety of arguments and the control paths that those arguments could trigger would make it infeasible to ensure that the program would follow all hooked rootkit code paths. By using programs that a rootkit tends to hide

[3]KQEMU is a host kernel module to enhance QEMU's performance by running some guest code natively on the host processor. It was disabled for the *SucKIT* experiments because it interferes with an assembly instruction related to the interrupt descriptor table.

[4]We choose this version of Linux because it allowed the most rootkits we experimented with to execute.

information from, we expect that at least a portion of the malicious rootkit code will be triggered. During the execution of those 10 utility programs, 39 different system calls get executed and those that led to rootkit code execution are shown in the "User-Level Impacts" column of the table. The last aspect of the profile is the extracted kernel rootkit code shown in the table only by the number of rootkit instructions extracted. This is useful for determining the approximate size of a kernel rootkit, and the code is made available by PoKeR for further analysis, as shown in Section 6.5.2.

6.5.1 Profiling-based Study of Rootkit Behavior

As a kernel rootkit investigation tool, PoKeR allows a human expert to quickly ascertain and classify a rootkit's attack methodology without solely relying on manual analysis of the rootkit's binary, source code, or the compromised OS. In the following, we summarize the findings that generalize across the rootkits we have profiled using PoKeR.

From the "hooking behavior" aspect, we can generalize the rootkits' profiles to three hooking strategies: modifying existing kernel code, hooking system call entries, and hooking function pointers in data structures. For example, one rootkit that we profiled, *SucKIT*, modifies existing kernel code. Five rootkits (*rial*, *rkit*, *knark*, *kbdv3*, and *adore* 0.42) use syscall hooking as their primary attack vector, with two others (*SucKIT* and *adore* 0.53) employing it in addition to other attack techniques. Two rootkits (*adore* 0.53 and *adore-ng* 0.56) hook function pointers in both static and dynamic kernel objects.

From the "targeted kernel objects" aspect, we can identify those kernel objects that are more likely to be manipulated by rootkits that manipulate kernel data structures directly. (This is also known as direct kernel object manipulation or DKOM). For example, some critical fields in the process control block (e.g., `uid`, `euid`) can be targeted (e.g., by the kbdv3 rootkit) for escalating the privilege of the process under which the rootkit code runs. The task list is often manipulated (e.g., by the *linuxfu*

and *hp* rootkits) for process hiding purposes. Moreover, the semantics associated with the function pointers hijacked by kernel rootkits also reveal the rootkits' intentions. For example, the function pointer `get_info` can be hijacked (e.g., by *adore* 0.53 and *adore-ng* 0.56) to point to a function used to filter out "sensitive" information so that a rootkit can remain invisible in the compromised system.

PoKeR's rootkit profiles can also reveal the changes made between various versions of the same rootkit. Consider the three different *adore* rootkits in Table 6.2. Version 0.42 relies solely on a system call hooking attack. A later version, 0.53, lessens its reliance on system call hooking and hooks two kernel objects instead. Once *adore* becomes *adore-ng*, it moves to entirely relying on hooks in kernel objects. Such an evolution of *adore*'s attack behavior is clearly illustrated by PoKeR's profiles.

6.5.2 Detailed Results for Three Representative Rootkits

When conducting in-depth analysis of kernel rootkits, PoKeR is especially helpful in providing a human expert with information related to *what* a kernel rootkit does so that the expert can more quickly understand it. In this section, we describe detailed profiling results for three kernel rootkits which each display different attack methodologies. Our intention is to show how a human expert can use PoKeR to quickly understand a rootkit's behavior without starting from the source code.

adore-ng 0.56

Hooking Behavior

The hooking profile for *adore-ng* reveals that it does not hook any system calls. In addition, one of its hooks requires combat tracking to reveal. The rootkit modifies six function pointers in various kernel objects. It modifies three function pointers in the `proc` file system. One of those pointers, `proc_net->(...)->get_info`, is located in an object that was dynamically allocated on the kernel's heap (and was found by combat tracking.) The other two, `proc_root_inode_operations->lookup` and

`proc_root_operations->readdir` are related to file operations on `proc`. The `proc` file system exports information from kernel-space to user-space and is used by applications that retrieve system information. `ps`, for example, retrieves the process list and `netstat` gets information about open network connections. Modifying function pointers in the `proc` filesystem allows the rootkit to hide processes and network connections. *Adore-ng* also impacts the main `ext3` file system. The first of these functions, `ext3_dir_operations->readdir`, is used to generate directory listings. The second function, `ext3_file_operations->write`, is used to write to files. The most obvious reason to hook `readdir` on the main file system would be to hide the existence of certain files. Lastly, *adore-ng* hijacks the `unix_dgram_ops->recvmsg` function pointer, which would allow it to intercept UNIX domain socket messages, a type of inter-process communication.

An analysis of the *adore-ng* source code reveals some additional observations that could not be obtained from only analyzing the profile. The `proc_root_inode_operations->lookup` hook is used to signal information to *adore-ng*'s kernel component from user-space. `ext3_file_operations->write` is modified to ensure that hidden processes do not write to any of the system wide log files in `/var`. `unix_dgram_ops->recvmsg` is changed to allow the rootkit to intercept and delete messages to the system logging daemon.

Targeted Kernel Objects

Based on PoKeR's profiling results, *adore-ng* does not modify any kernel objects outside of function pointers and instead does its work by hijacking the control flow. In this respect the rootkit is not any more advanced than many system call hooking rootkits. However, it is important to note that while it does not modify any other kernel objects, its malicious code may still be modifying the system call results returned to user-level programs.

User-Level Process Effects

Without modifying any system call table entries directly, *adore-ng* still manages to execute its malicious payload during system calls. This is logical, considering that the function pointers it modified would be called during various system calls. Our results show that five system calls from our corpus executed *adore-ng* code.

Extracted Code

Adore-ng results in 785 instructions extracted.

SucKIT 1.3b

Hooking Behavior

SucKIT only modifies one entry in the system call table, 59. This entry corresponds to a deprecated system call, `oldolduname`. A source code review reveals that *SucKIT* makes use of this system call entry to make the kernel function `kmalloc` callable from user-space. This allows it to allocate a place for its kernel component from user-space and then install it via `/dev/kmem`.

Targeted Kernel Objects

The targeted kernel objects lead to a few important observations. First, PoKeR's memory read log indicates that *SucKIT* reads the entire system call table. Second, it modifies the code of two kernel functions, `system_call` and `tracesys`. These two functions can be used to dispatch system calls. For example, when a software interrupt `0x80` is received, the `system_call` function directs the system call to the proper kernel handler by reading the function pointer from the system call table. These two observations indicate that *SucKIT* makes a copy of the system call table and modifies the dispatcher functions to use the new table instead of the old one.

User-Level Process Effects

In *SucKIT*'s profile, we observed no modifications to relevant function pointers other than `oldolduname`. However, because *SucKIT* directly overwrites kernel code in the

Table 6.3
Excerpt of SucKIT code extracted by PoKeR

Address	Order	Instruction
C72EC40B	22	`lcall 0x00000414`
C72EC410	-	
C72EC414	23	`pop %eax`
C72EC415	24	`ret`
...		
C72EE0CB	1	`push %ebp`
C72EE0CC	2	`mov %esp, %ebp`
C72EE0CE	3	`sub $0x0C, %esp`
C72EE0D1	4	`mov $0x00001000, %ecx`
C72EE0D6	5	`push %edi`
C72EE0D7	6	`push %esi`
C72EE0D8	7	`push %ebx`
C72EE0D9	8	`movl 0x14(%ebp), %eax`
C72EE0DC	9	`mov $0x0804EF39, %ebx`
C72EE0E1	10	`sub $0x0804D040, %ebx`
C72EE0E7	11	`movl 0xC(%ebp), %edx`
C72EE0EA	12	`movl %eax, 0xEC(%edx)`
C72EE0F0	13	`movl 0x8(%ebp), %esi`
C72EE0F3	14	`leal 0x400(%esi,%ebx), %esi`
C72EE0FA	15	`movl %esi, -0x4(%ebp)`
C72EE0FD	16	`mov $0x00, %dl`
C72EE0FF	17	`mov %esi, %edi`
C72EE101	18	`mov %dl, %al`
C72EE103	19	`repz stosb %al, %es:(%edi)`
C72EE105	21	`lcall 0xFFFFE40B`

Linux system call dispatcher, it still hijacks the control flow of key system calls using its alternate table. In our test suite, we find that *SucKIT* manages to hijack 10 of the 39 system calls.

Extracted Code

One portion of the extracted code was interesting enough to warrant inclusion here. Table 6.3 shows the first few dozen instructions executed by the *SucKIT* rootkit. The table shows the virtual address where the code was located, the order in which the

instructions were executed, and the extracted instructions themselves – all provided by PoKeR.

One unique property of *SucKIT* that can be seen from these instructions is the way it creates global variables. *SucKIT* installs itself into the kernel by writing its malicious kernel payload directly into a piece of memory specially `kmalloc`'d and then executing it. The specific address of kernel memory where *SucKIT* will reside is not known at compile time. Global variables (the addresses of which must be known at compile time) are not available to the rootkit author. Rootkits that install as kernel modules do not have this problem as the kernel will dynamically relocate their code and data prior to execution. Given that *SucKIT* does not have the benefit of dynamic relocation, a trick is used to permit the use of global variables when their addresses cannot be known a priori. Instruction 21 in the table (`lcall 0xFFFFE40B`) makes a function call to an offset of the current page, in this case a negative number. This call causes instruction 22 (near the top of the table) to execute. Starting at instruction 22 the layout is: instruction 22 followed by 4 bytes followed by instructions 23 and 24. When instruction 22 executes (another local call) the address of the memory immediately following the `lcall` is pushed onto the stack. This is the return address, but here it corresponds to the address of the 4 bytes of data. The `pop` instruction that runs next moves that address into register `eax` and then issues a `ret` that returns control flow back to the main code. At this point register `eax` contains the address of the 4 bytes of data. This mechanism allows the attacker to achieve the functionality of global variables without having to worry about dynamic relocation.

hide pid (hp) 1.0.0

Hooking Behavior

The *hp* rootkit modifies no function pointers and does not hijack control flow at all. It also does not install persistent code. This is different from the previous two rootkits.

Targeted Kernel Objects

The kernel object accessed by *hp* is the pid hash table. (`pidhash` is basically a table of task structures hashed by pid. It allows kernel functions to search for a process by pid without needing to traverse the entire process list. Entries in the hash table are still part of the process list, however.) It is possible to see the rootkit's functionality using the following excerpt from its object access log:

```
R - 0xc03a61a0 (0xc677c000): pidhash[600]
R - 0xc677c078 (0x0000025a): pidhash[600]->pid
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
R - 0xc677c050 (0xc76d8000): pidhash[600]->next_task
R - 0xc677c050 (0xc76d8000): pidhash[600]->next_task
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
W - 0xc76d8054 (0xc6780000): pidhash[600]->next_task->prev_task
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
W - 0xc6780050 (0xc76d8000): pidhash[600]->prev_task->next_task
```

As can be seen from the log, the rootkit reads `pidhash[600]` in the table, verifies it is the correct entry by checking the pid, and then proceeds to remove that entry from the process list by modifying the previous and next pointers of its neighbors. These task structures are dynamically allocated, yet our combat tracking technique is able to identify them accurately.

User-Level Process Effects

The *hp* rootkit did not execute any malicious code during the execution of our corpus.

Extracted Code

The extracted code of *hp* is small – only 100 instructions.

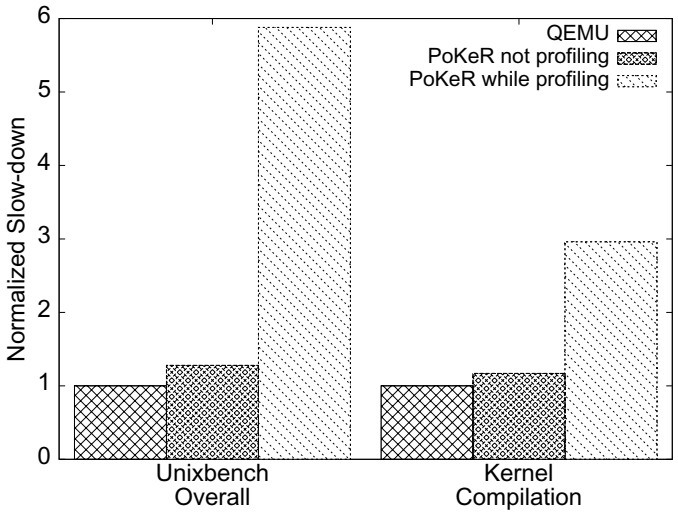


Figure 6.5. PoKeR performance results

Summary

The above analysis demonstrates that PoKeR's profiles are able to tell a human expert what a rootkit does to help her better understand the rootkit. The manual analysis of the source code frequently only provides further clarification to the human's interpretation of the PoKeR profile. Our experience while profiling these rootkits indicates that even when the rootkit source code is available, it is convenient to first use PoKeR – instead of starting from the source code – to achieve faster revelation and understanding of the rootkit's behavior.

6.6 Performance

While performance is not always a significant concern for a honeypot system, in this section we test the speed at which the various components of PoKeR run.

Runtime PoKeR Module

We ran two basic tests to determine the performance of PoKeR's runtime component that generates the log entries. All tests were run under the system as described at the beginning of Section 6.5. We ran Unixbench 4.1.0 as well as a test timing kernel compilation under standard QEMU, QEMU + PoKeR without a rootkit being profiled,

and QEMU + PoKeR while profiling the *adore-ng* rootkit. These two benchmarks were chosen because they test a variety of both CPU and I/O bound workloads. The results, normalized to the speed of standard QEMU, are shown in Figure 6.5. (Lower is better.) While in profiling mode, PoKeR is 2.96x slower than standard QEMU for the kernel compilation test and about 5.88x slower under the Unixbench test. While not profiling (but simply waiting to detect an attack), the slowdown is significantly less, 1.17x for the kernel compilation case and 1.28x for the Unixbench case.

QEMU

QEMU itself contributes a noticeable amount of overhead to our PoKeR prototype. Thoroughly benchmarking QEMU is outside the scope of this work, a basic benchmark is helpful for understanding PoKeR's overall performance. To test the overhead of QEMU we used version 0.9.1 (the latest release) and compared the performance of a native install of Ubuntu 8.10 to a QEMU+KQEMU virtualized copy. Both had access to the same amount of memory (512MB) and one processor core. A kernel compilation benchmark revealed an overhead of 3.8x. Given the portability of NICKLE to other dynamic translation-based VMMs such as VMware [91], we believe that this portion of overhead could be reduced by making use of a more efficient VMM platform.

Log Processing

To demonstrate the efficiency of log processing, the amount of time taken to process the log entries for each of the 10 rootkits in Tables 6.1 and 6.2 was measured. The longest processing time was for *rial*: 3 minutes and 36 seconds. The shortest time was for *rkit*: 0.7 second. The average time across all 10 rootkits was 37 seconds.

6.7 Discussion

In this section we will discuss potential attacks against PoKeR as well some of its limitations and future improvements.

6.7.1 Attacks

There exist a number of potential attacks that a rootkit may employ to evade PoKeR.

Our current prototype relies on NICKLE to signal the execution of kernel rootkit code. However, if a kernel rootkit modifies kernel data directly from user-space using a memory access device such as `/dev/kmem`, PoKeR will not be able to profile it. We have synthesized such a rootkit, although it has limited functionality as it cannot execute its own kernel code. A related attack is one that uses only existing kernel code as in an advanced type of return-to-libc attack [16,17] for the kernel. NICKLE would fail to generate the needed detection point for PoKeR. Existing approaches such as control-flow integrity [43] are able to detect these types of attacks and PoKeR could be engineered to use them to generate detection points.

Combat tracking implicitly relies on the assumption that a rootkit must obtain dynamic kernel objects' addresses by starting a chain of reads at a static data object. A rootkit may not need to do this, however. It may, for example, call existing kernel code to retrieve the address of a data structure. In this case, the chain of reads would occur from legitimate kernel code and hence would not be logged. PoKeR can handle this situation by simply tracking all kernel reads instead of only rootkit reads, but at an increased performance penalty. Another potential approach would be to have PoKeR monitor all kernel reads as long as there is a pointer to malicious code on the current kernel stack. This pointer is likely a return address to the rootkit code, which has called the valid kernel code.

Another situation is one where a rootkit installs a code hook and uses it to walk the stack and find kernel object addresses on it. (If the rootkit author knows what functions have already been called prior to his hook, he can easily derive the type information for function arguments on the stack.) In this case, combat tracking would not be able to properly identify the types of data being read. PoKeR could be

extended to monitor type information for items on the stack, similar to the way `gdb` does.

Finally, a rootkit may be able to scan kernel memory and guess at the identity of kernel objects, and do so with a high probability of success. One possible approach to combating this attack would be to periodically build a complete map of kernel objects (similar to SBCFI [47]). Assuming that this periodic map building occurred at regular intervals, PoKeR would be able to identify any dynamic kernel object with high probability, even without a chain of reads.

6.7.2 Limitations

There are some limitations to our current PoKeR prototype. First, our current profiling results are not complete for fully and provably determining all aspects of a given rootkit. Instead, we are only focusing on four specific aspects of the rootkit's behavior. Our lack of completeness is a trait shared by other dynamic analysis based systems [60, 121].

Second, the current prototype is still limited in revealing the context in which the rootkit-manipulated kernel objects were used. For example, in the `adore-ng` experiment we noticed that the IPC datagram receive function was hijacked. The derived profile, however, could not tell what this modification is able to accomplish. Manually inspecting the `adore-ng` source code indicated that this was used to filter messages being sent to `syslogd`. It would be advantageous if PoKeR could be improved to automatically reveal this. In the meantime, we also recognize that PoKeR's user-level impact metric is still simplistic and we would like to extend it to determine the complete set of system calls that may get hijacked at runtime. Correlating modified kernel objects with a static analysis of the kernel's call graph as well as multiple path exploration [121] are potential avenues of research in this area.

Finally, a rootkit may be able to detect virtualization or PoKeR's profiling mode and alter its actions accordingly. Note that as virtual machines become more preva-

lent, they are quickly becoming valid targets for attacks and rootkit authors are losing their incentive to avoid them. While efforts could be made to mask the presence of virtualization from the attacker, it is considered an unsolvable problem in the general sense [111].

6.8 Summary

In this chapter we demonstrated the applicability of the SMA to a security problem outside of code injection prevention, namely the profiling of injected code. We presented the design, implementation, and evaluation of PoKeR, a kernel rootkit profiler that produces multi-aspect rootkit profiles which include hooking behavior, targeted kernel objects, user-level impacts, and executed rootkit code. In particular, via the combat tracking technique, PoKeR maintains a map of dynamic kernel objects, which allows it to accurately determine which kernel objects are modified by a rootkit. PoKeR is also able to extract the executed rootkit code and infer the potential impact the rootkit might have on user-level programs. PoKeR was evaluated using 10 real-world kernel rootkits, the profiles of which reveal a variety of attack methodologies and demonstrate PoKeR's effectiveness as a rootkit analysis aid.

7 CONCLUSION

In this dissertation we have presented the split memory architecture, a memory architecture which is not susceptible to code injection attacks.

In Chapter 3 we discussed the basic design of this memory architecture and compared it to existing architectural approaches for the prevention of code injection attacks. We described why computer systems which employ the von Neumann memory architecture are susceptible to the attacks and discussed how the SMA addresses the problem by preventing the processor from accessing injected code during an instruction fetch. The SMA separates code and data into separate memory spaces and transparently redirects memory accesses accordingly. The end result is that programs and operating systems compiled for a von Neumann architecture are able to execute on an SMA without modification while still gaining the code injection immunity benefits. We also discussed trade-offs involved in using an SMA, specifically regarding increased memory usage and an inability to run self modifying code.

In Chapter 4 we demonstrated that an operating system can construct an SMA for individual processes which it hosts. We built our system in version 2.6.13 of the Linux kernel running on an Intel x86 processor and demonstrated its efficacy against the attacks in the Wilander benchmark as well as five exploits in real software. The system is able to prevent attacks that involve the execution of injected code, however it is unable to prevent return oriented programming attacks that make use of existing code. Given that return oriented programming has been demonstrated to be Turing complete in some instances [16], a solution to this limitation is one which should be pursued in future work. In terms of performance, our modified system was able to execute at higher than 80% of full speed for the four primary benchmarks we used.

In Chapter 5 we applied an SMA to prevent the execution of injected code by a running operating system kernel. Code injection is a tactic used by many kernel

rootkits. We built a VMM-based SMA called NICKLE for the purpose of preventing these rootkits. NICKLE was one of the first systems capable to preventing the execution of all unauthorized code at the kernel's privilege level. We implemented NICKLE in two different virtual machine monitors, QEMU and VirtualBox, and demonstrated the system's effectiveness against 22 different rootkits. One unique feature of NICKLE over other systems is its ability to protect memory pages containing both code and data. Both Linux and Windows contain these "mixed" pages, and as such it was vital that our implementation be able to protect them. In this work we did not thoroughly evaluate the reason for mixed pages in these two operating systems, however that would be an important next step in further evaluating the problem. Much like the SMA for the user-level, NICKLE is also limited to only preventing rootkit attacks that involve the execution of new code at the kernel's privilege level. The possibility still exists that attackers could make use of existing kernel code or modify only kernel data structures, however it is currently unknown how effective such attacks would be. In terms of performance, the QEMU port of NICKLE was able to execution at 99% of the speed of an unmodified version of QEMU.

In Chapter 6 we applied an SMA to the realm of behavioral profiling of injected code with PoKeR, a rootkit behavioral profiler based on NICKLE's VMM-base SMA. PoKeR is able to profile rootkits in a honeypot setting and reveal what a rootkit does to help a human expert understand it. PoKeR successfully demonstrates that the SMA has applications outside of code injection prevention. Because its separate code and data spaces allow it to identify injected code, the SMA opens up new opportunities in the realm of attack response through things like malicious code rewriting, forensic extraction of malicious code, and monitoring of malicious code execution.

7.1 Conclusions

There are a few conclusions that we have drawn based on our results.

- The SMA is extremely effective against code injection because it reveals and cuts off a root cause of code injection attacks: the processor's ability to fetch "data" for execution. Many existing techniques rely on preventing a specific control-flow hijacking methodology, which makes them unable to be used in new scenarios. For example, detecting modifications to a function return address may prevent a large number of attacks at the user-level, but the technique is not applicable to a kernel-level scenario where the attacker can load a malicious driver to inject code and modify the return address. The SMA is not tied to a specific control-flow hijacking technique, and as such is applicable to a large number of attack scenarios.

- Virtualizing one memory architecture on top of another needs be thought of from the perspective of cost versus benefit. The SMA provides a very strong benefit, the inability to execute injected code, but this comes at the cost of performance and the loss of some features such as self-modifying code. This means that the SMA cannot be used to fully protect programs that make use of just in time compilation [122], but it can protect many other programs. It is also important to look at the workload of the protected target. The user-level implementation of the SMA, for example, performs poorly under workloads with a high frequency of context switches, but performs very well for more general workloads. We have begun investigating a hybrid scheme that combines the SMA with the XD bit which may be able to significantly reduce the performance overhead.

7.2 Future Work

The code injection prevention work presented in this dissertation, especially when applied to kernel rootkits, has opened up new opportunities for both offensive and defensive future work.

- **Virtualization as an OS Security Extension** Hardware virtualization (such as Intel's VT and AMD's SVM) is advancing and reducing the performance penalty of using such features to almost zero. This support is currently being used to produce VM hypervisors, but we are interested in leveraging the elevated privilege level of these processors to create a small operating system monitor capable of protecting the operating system from malicious tampering or unintended faults. By keeping the monitor small and only concerned about security, the overhead of such an approach can be negligible and provide a smaller, less complicated code base to be targeted for attack. We are particularly interested in finding new protection schemes that go beyond the current work of preventing unauthorized code execution.

- **Data-only Kernel Rootkit Attacks** Now that NICKLE is able to effectively prevent the execution of code injected into an OS kernel, rootkit attacks will need to shift to other attack vectors. We will further research attack and defense techniques related to rootkit attacks that operate by directly modifying kernel data structures but not executing any new code in the kernel. Thus far there is little existing work constructing or defending against these data-only attacks. This work is challenging because many kernel data structures are frequently changed during normal usage, making it difficult to distinguish between valid and invalid states.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Eugene H. Spafford. The Internet Worm Program: An Analysis. *SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[2] James P. Anderson. Computer Security Technology Planning Study, Volume I. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscom Air Force Base, Bedford, MA, October 1972. See page 61.

[3] J. von Neumann. First Draft of a Report on the EDVAC. 1945. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 355–364, 1975.

[4] H. H. Aiken. Proposed Automatic Calculating Machine. 1937. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 191–198, 1975.

[5] H. H. Aiken and G. M. Hopper. The Automatic Sequence Controlled Calculator. 1946. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 199–218, 1975.

[6] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[7] James P. Anderson Co. Computer Security Threat Monitoring and Surveillance. Technical Report Contract 79F296400, February 1980.

[8] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[9] R. A. Meyer and L. H. Seawright. A Virtual Machine Time-sharing System. *IBM Systems Journal*, 9(3):199, 1970.

[10] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[11] William Stallings. *Operating Systems Internals and Design Principles*. Prentice Hall, Upper Saddle River, New Jersey 07458, Fourth edition, 2001.

[12] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., Sixth edition, 2002.

[13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 1997.

[14] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49). Article 14.

[15] Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack*, 11(58). Article 4.

[16] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (On the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[17] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[18] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravi Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[19] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[20] David Endler. The Evolution of Cross-site Scripting Attacks. Technical report, iDEFENSE Labs, May 2002.

[21] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.

[22] H. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. *http://www.trl.ibm.com/projects/security/ssp/*. Accessed December 2006.

[23] Crispin Cowan, P. Wagle, C. Pu, Steve Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of DARPA Information Survivability Conference and Expo*, 1999.

[24] Vendicator. Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux. *http://www.angelfire.com/sk/stackshield/info.html*. Accessed December 2006.

[25] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 13–26, May 1997.

[26] Lap-chung Lam and Tzi-cker Chiueh. Checking Array Bound Violation Using Segmentation Hardware. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 388–397, 2005.

[27] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[28] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[29] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* Intel Corporation, 2006. Publication number 253668.

[30] A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. *http://support.microsoft.com/kb/875352*. Accessed December 2006.

[31] PAX PAGEEXEC Documentation. *http://pax.grsecurity.net/docs/pageexec.txt*. Accessed May 2009.

[32] PAX SEGMEXEC Documentation. *http://pax.grsecurity.net/docs/segmexec.txt*. Accessed May 2009.

[33] Buffer Overflow Attacks Bypassing DEP (NX/XD bits) - Part 2 : Code Injection. *http://www.mastropaolo.com/?p=13*. Accessed December 2006.

[34] Paul Williams and Eugene H. Spafford. An Exploration of Highly Focused, Coprocessor-based Information System Protection. *Computer Networks*, 51(5):1284–1298, April 2007.

[35] Babak Salamat, Andreas Gal, and Michael Franz. Reverse Stack Execution in a Multi-variant Execution Environment. In *Proceeding of the Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[36] PAX ASLR Documentation. *http://pax.grsecurity.net/docs/aslr.txt*. Accessed December 2006.

[37] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[38] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[39] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems*, October 2003.

[40] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[41] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[42] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[43] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[44] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[45] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 239–242, 2002.

[46] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[47] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[48] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 368–377, 2005.

[49] Xuxian Jiang, Xinyuan Wang, and Dongyan 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*, October 2007.

[50] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[51] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2007.

[52] Simson Garfinkel and Gene Spafford. *Web Security & Commerce*, chapter 9. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1997.

[53] Microsoft. Driver Signing for Windows. *http://technet.microsoft.com/en-us/library/cc784714.aspx*. Accessed November 2008.

[54] Fred Cohen. A Cryptographic Checksum for Integrity Protection. *Computers and Security*, 6(6):505–510, December 1987.

[55] Fred Cohen. Models of Practical Defenses Against Computer Viruses. *Computers and Security*, 8(2):149–160, April 1989.

[56] Jeffrey Wilhelm and Tzi-cker Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of Recent Advances in Intrusion Detection*, pages 219–235, September 2007.

[57] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 91–100, 2004.

[58] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, 2007.

[59] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, July 2008.

[60] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, February 2009.

[61] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, February 2008.

[62] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In *Proceedings of Recent Advances in Intrusion Detection*, pages 21–38, September 2008.

[63] Atmel Corporation. *8-bit AVR®Microcontroller with 128K Bytes In-System Programmable Flash*. 2008. Document 2467.

[64] Wind River: VxWorks. *http://www.windriver.com/vxworks/*. Accessed March 2007.

[65] Anatol W. Holt. Program Organization and Record Keeping for Dynamic Storage Allocation. *Communications of the ACM*, 4(10):422–431, 1961.

[66] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, 1972.

[67] Alastair J. W. Mayer. The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times? *SIGARCH Computer Architecture News*, 10(4):3–10, 1982.

[68] Intel Corporation. *INTEL 80386 Programmer's Reference Manual*. Intel Corporation, 1987.

[69] John Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Communications of the ACM*, 4(10):435–436, 1961.

[70] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An Architectural Approach to Preventing Code Injection Attacks. In *Proceedings of the 37th Annual International Conference on Dependable Systems and Networks*, pages 30–40, 2007.

[71] kernelthread.com: Securing Memory. *http://www.kernelthread.com/publications/security/smemory.html*. Accessed December 2006.

[72] Pedro Venda. PaX Performance Impact. *http://www.pjvenda.org/linux/doc/pax-performance/*, October 2005. Accessed May 2009.

[73] Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted Circumvention of Self-Hashing Software Tamper Resistance. *IEEE Transactions Dependable and Secure Computing*, 2(2):82–92, 2005.

[74] Sebek. *http://www.honeynet.org/tools/sebek/*. Accessed May 2009.

[75] Xuxian Jiang and Xinyuan Wang. "Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots. In *Proceedings of Recent Advances in Intrusion Detection*, pages 198–218, September 2007.

[76] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. In *Proceedings of the 1st ACM European Conference on Computer Systems*, pages 15–27, 2006.

[77] Robert C. Daley and Jack B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, 1968.

[78] Axelle Apvrille, David Gordon, Serge Hallyn, Makan Pourzandi, and Vincent Roy. DigSig: Run-time Authentication of Binaries at Kernel Level. In *Proceedings of the 18th USENIX Conference on System Administration (LISA)*, pages 59–66, 2004.

[79] Brett Lymn. Verified Exec – Extending the Security Perimeter. In *Australian Unix Users Group Conference*, 2004.

[80] Solar Eclipse. openssl-too-open. *http://www.phreedom.org/solar/exploits/apache-openssl/*. Accessed June 2009.

[81] lsd-pl.net. BIND 8.2.x (TSIG) Remote Root Stack Overflow Exploit (2). *http://milw0rm.org/exploits/279*. Accessed June 2009.

[82] CERT. CERT® Incident Note IN-2001-03, Exploitation of BIND Vulnerabilities. *http://www.cert.org/incident_notes/IN-2001-03.html*. Accessed May 2009.

[83] Solar Eclipse. proftpd-not-pro-enough. *http://www.phreedom.org/solar/exploits/proftpd-ascii/*. Accessed June 2009.

[84] eSDee. samba-2.2.8 < remote root exploit. *http://downloads.securityfocus.com/vulnerabilities/exploits/sambal.c*. Accessed June 2009.

[85] TESO Security. 7350wurm – x86/linux wu_ftpd remote root exploit. *http://examples.oreilly.com/networksa/tools/7350wurm.c*. Accessed June 2009.

[86] The Apache HTTP Server Project. *http://httpd.apache.org/*. Accessed December 2006.

[87] Linux/Unix nbench. *http://www.tux.org/~mayer/linux/bmark.html*. Accessed December 2006.

[88] Unixbench. *http://www.tux.org/pub/tux/benchmarks/System/unixbench/*. Accessed December 2006.

[89] Digital Equipment Corporation. *KB11-A Central Processor Unit Maintenance Manual*. 1972. Section 4.3.4.2 – Access Modes.

[90] Jonathon Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening Software Self-Checksumming via Self-Modifying Code. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 18–27, December 2005.

[91] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of Recent Advances in Intrusion Detection*, pages 1–20, September 2008.

[92] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[93] Innotek. Virtualbox. *http://www.virtualbox.org/*. Accessed September 2007.

[94] Intel Corportation. Virtualization Technologies from Intel. *http://www.intel.com/technology/virtualization/*. Accessed June 2009.

[95] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.12 edition*. September 2006.

[96] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[97] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2003.

[98] 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 ACM Symposium on Operating Systems Principles*, pages 91–104, 2005.

[99] James P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscom Air Force Base, Bedford, MA, October 1972.

[100] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[101] sd and devik. Linux on-the-fly Kernel Patching without LKM. *Phrack*, 11(58). Article 7.

[102] fuzen_op. FU Rootkit. *http://www.rootkit.com/project.php?id=12*. Accessed September 2007.

[103] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[104] Julian B. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, May 2006.

[105] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[106] Tobias Klein. Scooby Doo – VMware Fingerprint Suite. *http://www.trapkit.de/research/vmm/scoopydoo/index.html*. Accessed June 2009.

[107] Joanna Rutkowska. Red Pill: Detect VMM Using (Almost) One CPU Instruction. *http://invisiblethings.org/papers/redpill.html*, November 2004.

[108] F-Secure Corporation. Agobot. *http://www.f-secure.com/v-descs/agobot.shtml*. Accessed June 2009.

[109] Kostya Kortchinsky. Honeypots: Counter Measures to VMware Fingerprinting. *http://seclists.org/honeypots/2004/q1/0015.html*, January 2004. Accessed June 2009.

[110] Tom Liston and Ed Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. *http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf*, 2006. Accessed June 2009.

[111] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[112] Advanced Micro Devices. *Whitepaper: AMD-V^{TM} Nested Paging*. July 2008.

[113] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 47–60, April 2009.

[114] Greg Hoglund. Kernel Object Hooking Rootkits (KOH Rootkits). *http://www.rootkit.com/newsread.php?newsid=501*, 2006. Accessed November 2008.

[115] Peter Silberman and C.H.A.O.S. FUTo. *Uninformed*, 3, 2006. `http://uninformed.org/?v=3&a=7&t=sumry`.

[116] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, December 2007.

[117] Hans Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software, Practice and Experience*, 18(9):807–820, September 1988.

[118] libdisasm. x86 Disassembler Library. *http://bastard.sourceforge.net/libdisasm.html*. Accessed September 2008.

[119] Free Software Foundation. GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/*. Accessed October 2008.

[120] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 255–266, December 2008.

[121] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[122] Andreas Krall. Efficient JavaVM Just-in-time Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, 1998.

VITA

VITA

Ryan Riley entered Purdue University in the Fall of 2000 and received a B.S. in computer engineering in May 2004. He went on to obtain an M.S. in computer science in May 2006 and a Ph.D. in August 2009 under the direction of Professor Dongyan Xu and Professor Xuxian Jiang. During his time in graduate school he was awarded the Purdue University Graduate Student Award For Outstanding Teaching, the Best Paper Award from the 2008 Recent Advances in Intrusion Detection conference, and the CERIAS Diamond Award For Outstanding Academic Achievement. In the Fall of 2009 he joined the faculty of Qatar University as an Assistant Professor of Computer Science and Engineering.