Improved Kernel Security Through Code Validation, Diversification, and Minimization

by Dannie M. Stanley
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

IMPROVED KERNEL SECURITY THROUGH CODE VALIDATION, DIVERSIFICATION, AND MINIMIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Dannie M. Stanley

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

To my wife, my parents, and my children.

ACKNOWLEDGMENTS

I would like to express my deep appreciation and gratitude to my advisors Dr. Eugene H. Spafford and Dr. Dongyan Xu for their patience, guidance, encouragement and the many wonderful opportunities that they afforded me. I am fortunate to have the opportunity to learn from such highly respected and accomplished mentors. I would also like to thank my additional committee members Dr. Sonia Fahmy and Dr. Samuel Liles for their guidance and helpful feedback.

I would also like to thank Dr. Ryan Riley. Ryan has been a peer-mentor to me. He helped guide me through the early years of graduate school. I am forever indebted to him and aspire to follow in his footsteps.

Finally, I would like to thank my wife. She is my champion. She provided light when the rigors of graduate school were casting a shadow. She has sacrificed beyond what I wished for her and has done so with grace.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Stanley, Dannie M. Ph.D., Purdue University, December 2013. Improved Kernel Security Through Code Validation, Diversification, and Minimization. Major Professors: Eugene H. Spafford and Dongyan Xu.

The vast majority of hosts on the Internet, including mobile clients, are running one of three commodity, general-purpose operating system families. In such operating systems the kernel software executes at the highest processor privilege level. If an adversary is able to hijack the kernel software then by extension he has full control of the system. This control includes the ability to disable protection mechanisms and hide evidence of compromise.

The lack of diversity in commodity, general-purpose operating systems enables attackers to craft a single kernel exploit that has the potential to infect millions of hosts. If enough variants of the vulnerable software exist, then mass exploitation is much more difficult to achieve. We introduce novel kernel diversification techniques to improve kernel security.

Many modern kernels are self-patching; they modify themselves at run-time. Self-patching kernels must therefore allow kernel code to be modified at run-time. To prevent code injection attacks, some operating systems and security mechanisms enforce a $W \oplus X$ memory protection policy for kernel code. This protection policy prevents self-patching kernels from applying patches at run-time. We introduce a novel run-time kernel instruction-level validation technique to validate the integrity of patches at run-time.

Kernels shipped with general-purpose operating systems often contain extraneous code. The code may contain exploitable vulnerabilities or may be pieced together using return/jump-oriented programming to attack the system. Code-injection prevention techniques do not prevent such attacks. We introduce a novel run-time kernel minimization technique to improve kernel security.

We show that it is possible to strengthen the defenses of commodity general-purpose computer operating systems by increasing the diversity of, validating the integrity of, and ensuring the minimality of the included kernel components without modifying the kernel source code. Such protections can therefore be added to existing widely-used unmodified operating systems to prevent malicious software from executing in supervisor mode.

1 INTRODUCTION

1.1 Background

The vast majority of hosts on the Internet, including mobile clients, are running one of three commodity, general-purpose operating system families. In such operating systems the kernel software executes at the highest processor privilege level and therefore has full control of the system. If an adversary is able to hijack the kernel software then by extension he has full control of the system. This control includes the ability to disable protection mechanisms and hide evidence of compromise. Attackers may exploit flaws in software to subjugate the kernel using a variety of attack techniques, including buffer overflows [1], return-oriented programming [2, 3], heap overflows [4], format string vulnerabilities [5], and integer overflows [6, 7].

Malicious operating system kernel software, such as the code introduced by a kernel rootkit, is strongly dependent on the organization of the victim operating system. The lack of diversity in commodity, general-purpose operating systems enables attackers to craft a single kernel exploit that has the potential to infect millions of hosts. If the underlying structure of vulnerable operating system components has been changed in an unpredictable manner, then attackers must create many unique variations of their exploit to attack vulnerable systems en masse. If enough variants of the vulnerable software exist, then mass exploitation is much more difficult to achieve. Therefore, *diversification* can be used to improve the security of an operating system kernel.

Security mechanisms have been created to prevent kernel rootkit code injection by authenticating all code that is loaded into kernel space. Modern self-patching kernels modify kernel code in-memory at run-time at instruction-level granularity. If the kernel code was validated at load-time and modified later at run-time, then the code is no longer guaranteed authentic. Instruction-level run-time *validation* can be used to improve the security of an operating system kernel.

Kernels shipped with general-purpose operating systems often contain extraneous code. The unnecessary kernel code is a security liability. The code may contain exploitable vulnerabilities or may be pieced together using return/jump-oriented programming to attack the system. Run-time kernel *minimization* can be used to improve the security of an operating system kernel.

Our hypothesis is the following: *It is possible to strengthen the defenses of commodity, general-purpose computer operating systems by increasing the diversity of, validating the integrity of, and ensuring the minimality of the included kernel components without modifying the kernel source code. Such protections can therefore be added to existing, widely-used, unmodified operating systems to prevent malicious software from executing in supervisor mode.*

To test our hypothesis we design and implement six distinct kernel security mechanisms, protect many unmodified commodity operating systems kernels using the mechanisms, and assail the protected kernels using common attack techniques including return-oriented programming and kernel rootkits.

1.2 Contributions

1.2.1 Diversification

Many forms of automatic software diversification have been explored and found to be useful for preventing malicious software infection. Forrest et. al. make a strong case for software diversity and describe a few possible techniques including adding or removing nonfunctional code, reordering code, and reordering memory layouts [8]. Our diversification techniques build on the latter.

We describe the design and implementation of two novel kernel diversification mechanisms that mutate an operating system kernel using memory layout randomization. We introduce a new method for randomizing the stack layout of function arguments and refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record.

We developed prototypes of our kernel diversification mechanisms using the plugin architecture offered by GCC. To test the security benefits of our techniques, we randomized multiple Linux kernels using our compiler plugins. We attacked the randomized kernels using multiple kernel rootkits. We show that by strategically selecting just a few components for randomization, our techniques prevent kernel rootkit infection.

1.2.2 Validation

Previous works in code injection prevention use cryptographic hashes to validate the integrity of kernel code at load-time. The hash creation and validation procedure depends on immutable kernel code. However, some modern kernels contain self-patching kernel code [9]; they may overwrite executable instructions in memory *after* load-time. Such dynamic patching may occur for a variety of reason including CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging. Previous hash-based validation procedures cannot handle such modifications.

We describe the design and implementation of a novel kernel validation mechanism that validates the integrity of each modified instruction as it is introduced into the guest kernel.

We developed prototypes of our kernel validation mechanism by customizing the Linux KVM hypervisor and performed instruction-level validation of a guest self-patching Linux kernel. Our experiments show that our system can correctly detect and validate all valid instruction modifications and reject all invalid ones in support of a new code-injection prevention system named NICKLE-KVM.

1.2.3 Minimization

Previous work has been done in additive operating system specialization [10]. An example of such specialization is loadable kernel modules, such as drivers, that add kernel code at runtime. However, work has not been done in subtractive operating system specialization.

We describe the design and implementation of two novel run-time kernel minimization mechanisms. The first is an out-of-the-box function eviction technique. The second is a kernel-based non-executable page technique.

We developed prototypes of our kernel minimization mechanism by customizing the Linux KVM hypervisor and performed run-time kernel minimization on a guest Linux kernel. Our experiments show that it is possible to improve the security of a kernel against return and jump oriented programming attacks by deactivating extraneous kernel code at run-time thereby limiting the supply of reusable instructions that can be used to construct return-oriented gadgets.

1.3 Terminology

Commodity General-Purpose Operating Systems Operating systems that are commonly used with desktop, laptop, and server-class hardware including the following families of operating system: Linux, Windows, and Mac OS X. This class of operating system excludes specialized operating systems used for embedded devices and mobile computing.

1.4 Models

1.4.1 Computation Model

Our computation model follows the Von Neumann architecture described by [11], used by many modern architectures [12–14], and illustrated in Figure 1.1. The computation model has four primary components: the control unit (CU), the arithmetic

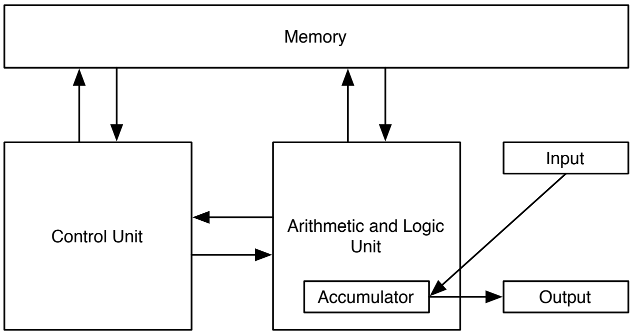


Figure 1.1. Computation model

and logic unit (ALU), accumulator that manages input/output operations, and memory. The CU has many registers including one used as the program counter (PC) that are used to fetch and execute instructions as described in Section 1.4.2. The address of the next instruction to be executed is stored in the PC. The memory holds both code and data as described in Section 1.4.3. This model of computation is sometimes referred to as the "Princeton" architecture in contrast to the "Harvard" architecture that has distinct physically separate memory for code and data. The ALU is responsible for performing arithmetic and logical calculations.

Further we assume a linearly addressed hardware-assisted paged virtual memory system with a memory management unit (MMU). A special processor control register[1] holds the memory address of the page directory base register (PDBR). The PDBR address is the entry point for the page table associated with the current task. Upon a context switch the PDBR may be updated giving each task its own linearly addressed virtual memory space. The MMU also utilizes a translation look-aside buffer[2] (TLB). We assume that the TLB is flushed when the PDBR is updated. Further in our model, virtual memory pages have read-write-execute access control attributes that can be used for page-level protections.

[1] CR3 for x86

[2] for simplicity a unified TLB is assumed though some modern processors have multiple TLBs

1.4.2 Execution Model

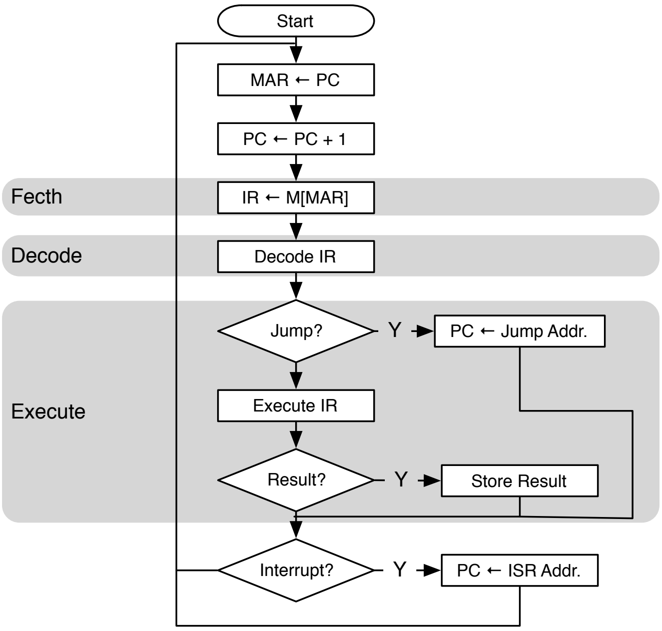


Figure 1.2. Execution model

Our execution model follows the fetch-decode-execute design described by [15], used by many modern architectures [12–14], and illustrated in Figure 1.2. The execution model has three phases during one clock cycle: fetch, decode, and execute. The PC holds the memory address of the next executable instruction. At startup, the CPU sets an initial PC value. The CPU copies the PC value to the memory address register (MAR) and advances the PC to the next instruction. Then the CU *fetch*es the instruction from memory (M[]), using the address stored in MAR, into the instruction register (IR). Then the CU *decode*s the instruction into its component parts including the operation and operands. If the operation is a jump, then the PC is updated and the execution continues. Otherwise the ALU *execute*s the instruction and stores the result if one is emitted. If the interrupt request register (IRR) holds a pending interrupt, the programmable interrupt controller (PIC) copies the address of the corresponding interrupt service routine (ISR) to the PC.

For our model, we assume a single processing core running in a multitasking operating system. As a result, only one instruction is executed at a time and the operating system performs frequent context switches. The processor has at least two privilege modes that we refer to as kernel mode and user mode. Privileged instructions are only executed in kernel mode. Examples of such instructions includes: loading the interrupt descriptor table and changing the current privilege level. None of the techniques describe depend upon a specific task scheduling technique, instruction pipelining, instruction caching, or multi-core processing. Therefore, these components are not pertinent to our execution model.

1.4.3 Process Memory Model

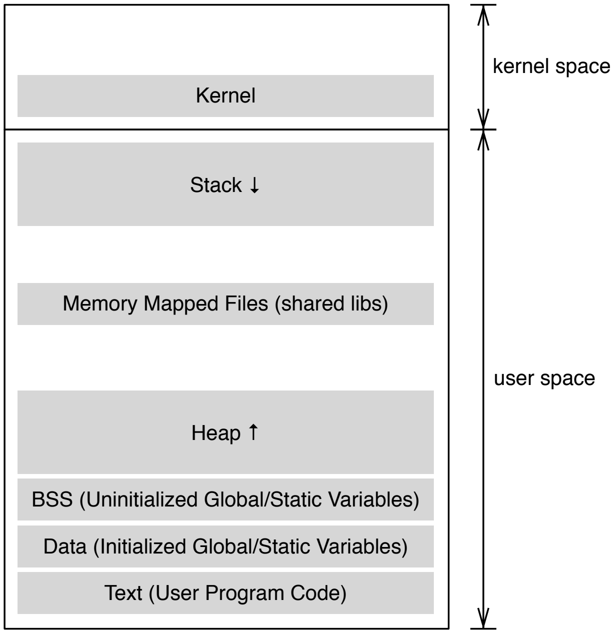

Kernel
kernel space
Stack ↓
Memory Mapped Files (shared libs)
user space
Heap ↑
BSS (Uninitialized Global/Static Variables)
Data (Initialized Global/Static Variables)
Text (User Program Code)

Figure 1.3. Task memory model

Our process memory model follows the memory layout used by many current commodity operating systems[3], and illustrated in Figure 1.3. The user program code

[3] Linux, Windows, and Mac OS

is loaded at the bottom of the process virtual address space. Dynamic user memory is allocated on the user heap (heap) adjacent to the user program, and it grows from a lower numerical memory address to a higher numerical memory address. The stack is located adjacent to the kernel space, described next, and grows from a higher numerical memory address to a lower numerical memory address. Shared libraries and other special files are mapped into a region of memory between the heap and the stack. These regions of memory are referred to collectively as "user space." A predetermined fixed amount of memory is allocated for the operating system "kernel space" at the top of the task address range. The kernel program is loaded at the bottom of the kernel space. Dynamic kernel memory is allocated as needed in available kernel space. Though the operating system may have one or more kernel stacks, unless otherwise noted the term "stack" refers to the user stack.

Generally, we rely on the stack-based calling convention design described in [16]. Specifically, the function calling convention used, unless otherwise noted, is the x86 CDECL calling convention described by Kernighan and Ritchie [17]. This calling convention is the default for current popular x86 C compilers[4]. This specific calling convention, though implementation specific, is germane to the attack and protection techniques described herein because of assumptions made about the layout of the stack during run-time.

In the CDECL calling convention, the calling function pushes the arguments for the callee function onto the stack in reversed order and then calls the callee function. When the call instruction is executed the current PC is pushed onto the stack, this memory address value is referred to as the "return address." The callee pushes the stack base pointer (BP) onto the stack and stores the current stack pointer (SP) as the new BP. The SP always holds the address of the top of the stack. The values resident in any of the registers that are needed by the function are pushed onto the stack. Then the callee allocates space for local, non-static, function variables on the stack by advancing the SP address proportional to the amount of space needed by local

[4] GNU GCC 4.7.2 and Microsoft Visual Studio 2012

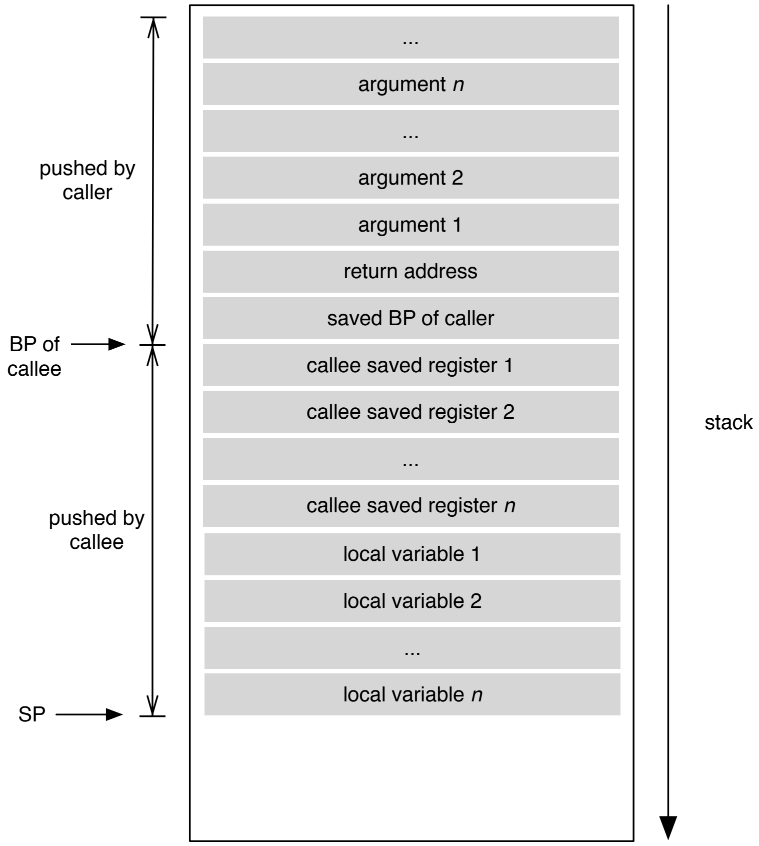


Figure 1.4. Call stack model

variables. Figure 1.4 illustrates the layout of the stack after a call has been made. The callee is responsible for restoring the callee saved registers before returning. When a return instruction is reached, the return address is popped off the top of the stack into the PC.

Many of the techniques described are not necessarily dependent on this task memory model and have analogues in other platforms and calling conventions. However, for the sake of clarity and harmony with related works, we provide examples using these conventions.

1.4.4 Threat Model

For our work we assume that an attacker can communicate with the vulnerable system through a network interface controller (NIC); that the system has a hardware or software vulnerability that is exploitable through NIC communications, such as a system service buffer overflow vulnerability; that the attacker is able to write to some region of memory, such as the user stack; and that the attacker is able to manipulate the PC either directly or indirectly. We assume that the operating system attempts to enforce contemporary protection mechanisms such as those listed in Section 2.2. Specifically, we assume that kernel code pages are read-only but kernel data pages are writeable and non-executable as implemented by the kernel or by VMM based solutions such as NICKLE [18].

2 RELATED WORK

2.1 Attack Techniques

The operating system kernel may be subverted using a variety of attack techniques. One of the primary types of attacks is referred to as *control-flow hijacking.* In this type of attack, the control flow of a running program is diverted to an unintended segment of code. This is accomplished by modifying an address that is stored in memory and will be later used by a control flow instruction such as a jump, call or return instruction. For example, in the x86 architecture, when a "ret" instruction is encountered, an address is popped off the stack into the program counter. This address is referred to as the "return address." As a result, the next instructions executed by the processor will be fetched from the memory region starting at the return address. Similarly, if a program makes a call to an address stored in data memory, such is the case when function pointers are used, and that address can be modified by an attacker, then the control flow may be subverted.

In this section we discuss the various methods used to leverage a control-flow type of attack. For each attack, the naive approach is described. In some cases mitigation techniques have been adopted that require the attack technique to be more sophisticated than the one presented. Mitigation techniques are described in Section 2.2.

2.1.1 Buffer Overflow Attack

Buffer overflow attacks are well studied but have "been remarkably resistant to elimination" [19]. Attackers have used buffer overflows in combination with other techniques, as we describe later, to hijack the control flow of a program and execute malicious code. Buffer overflow attacks can take place on the memory stack or the memory heap. Following is a description of both kinds of buffer overflow attacks and how they can lead to control-flow hijacking.

Stack-Based Buffer Overflow Attack

A piece of software is vulnerable to a stack-based buffer overflow attack if the boundaries of a buffer in memory allocated on the stack, such as a variable representing an array, are not checked and enforced when the buffer is filled. For example, a programmer may designate a local variable of the size 32 bytes. When the procedure holding this local variable is executed, the function preamble allocates a chunk of memory on the stack for this variable in the size of 32 bytes. If during program execution another buffer, larger than 32 bytes, is copied into this local variable then the space on the stack allocated for this variable is *overflowed*.

The ramifications of an overflow event may not be immediately apparent. The problem arises when the areas of memory adjacent to the buffer, such as other local variables, get overflowed *into*. Now those variables have values that were not intentionally set by the programmer.

In an execution model where the return address is stored on the stack and the stack grows downward, the top of the stack is at a lower numerical address, the function return address is at a higher numerical address than the local variable holding the buffer. As a result, a carefully crafted overflow event can modify the return address of the function. When the function concludes, the control flow jumps to the return address stored on the stack even if address has been malicious manipulated. This kind of attack is often referred to as “stack smashing” [1]. This technique was used by the first Internet worm described by Spafford in 1989 [20].

Heap-Based Buffer Overflow Attack

Similar to the stack-based buffer overflow attack, a piece of software is vulnerable to a heap-based buffer overflow attack if the boundaries of a buffer in memory allocated on the heap are not checked and enforced when the buffer is filled. Generally, a heap-based buffer overflow attack can be used to write arbitrary data into arbitrary locations in memory. As a result, a heap based overflow attack, could also modify the return address stored on the stack.

Similar to the stack-based attack the key to the heap-based attack is what gets written into memory when the buffer is overflowed. Most of the buffer overflow attacks that can be leveraged on the stack can also be leveraged on the heap. However, the heap does have a characteristic that distinguishes it from the stack-based vulnerabilities. When memory is allocated on the heap, bookkeeping information about the allocation itself is often stored alongside the memory chunks being allocated. Therefore, when an overflow event occurs, the bookkeeping information may get manipulated. Suppose for example, that the bookkeeping information contains linked-list like pointers to the next and previous blocks of allocated memory. When a block of memory is deallocated the bookkeeping information stored on adjacent linked list nodes is updated. Depending on the specific implementation of the allocation routines, the update can be be tricked into writing a value from the next pointer, e.g. a new return address, into an address pointed to by the previous pointer, e.g. the location of the existing return address [4]. Other heap overflow techniques focus on manipulating data stored in the heap, such as function pointers, to hijack control flow. Modifying a function pointer is not as general of a technique as modifying the return address because it is heavily dependent on the organization of the data structures used by the program.

2.1.2 Format String Attack

A piece of software is vulnerable to a format string attack if the format parameter to any of the functions in the "printf" family of library function can be provided by the attacker [5]. There are two important attributes of the format string functions that make this attack possible. First, the number of parameters passed into the printf functions is variable. Second, the formatting character sequence "%n" in the format string writes the number of outputted bytes to the corresponding integer pointer also provided as a parameter. Because the number of parameters are variable, the printf functions pop values off the stack to satisfy the formatting string. For example, if the formatting string contains placeholders for 10 integers, then 10 integers are popped off the stack. Even if 10 integers are not provided to the function, 10 integers are still popped off the stack. If one of the placeholders is a "%n", then the corresponding integer popped off the stack is treated as an integer pointer and it receives the count value. The count value can be artificially incremented using special formatting parameters. The count value can therefore be set to any integer value, for example a memory address. The destination of the write is determined by the value popped off the stack. The format string itself is stored on the stack. As a result, if the prefix of the format string is a sequence of non-null bytes, they can be interpreted as an integer, for example the address of the return address. As a result, any integer can be written to any memory address if the attacker is able to control the format string parameter.

2.1.3 Integer Overflow Attack

Unlike the previous attacks, integer overflow attacks do not themselves lead to control-flow hijacking. Instead they are used in combination with buffer overflow attacks. An integer overflow occurs when a value is stored in a variable of insufficient size. This can happen as a result of a calculation. For example, adding two large integers together may result in a number that is larger than the largest possible integer. Therefore, the sum should not be stored in an integer variable.

This can also happen as a result of a condition called “integer promotion.” When a calculation involves variables of two different sizes, then the smaller variable is promoted to the size of the larger variable, then demoted at the conclusion of the calculation. In both cases, the value stored in a variable that has been overflowed, is unintentionally allowed by the programmer.

Both overflow and integer promotion may lead to logic problems later in the code. For example, the overflowed variable may be used to check the boundaries of a buffer. If an attacker can force an integer overflow, he may be able to cause a buffer overflow [6, 7].

2.1.4 Return and Jump-Oriented Programming

The attacks previously mentioned all hijack the control-flow of a program. Generally they overwrite some area of memory that holds an address such as the return address on the stack. At some point during program execution, that address is used as the target of a jump, call or return instruction. At the location of the target memory address is a sequence of instructions that represent the attack code.

The attack code can be injected using one of the previous attack techniques, for example a buffer overflow. However, various mitigation strategies, as described in Section 2.2 have been introduced to prevent the injection of code into executable regions of memory or to prevent the execution of writable area of memory such as the regions occupied by heap and the stack. As a result, a different kind of attack evolved that reuses portions of existing authorized code, such as kernel and system libraries, in unconventional combinations to carry out the attack rather than injecting new code. These kinds of attacks are generally referred to as “return-oriented programming” attacks. Following is a description of such attacks.

Return-Into-Libc Attack

A return-into-libc attack was first described in 1997 as a proof-of-concept attack against a system with a non-executable stack [21]. If an attacker can write data to the stack, by using a buffer overflow for example, the stack can be strategically crafted so that the program execution jumps into a shared library function. The function executes exactly as if it had been called conventionally including popping function parameters off the stack[1]. Therefore, both the function and function parameters can be provided by the attacker.

By way of an example, the attack described in [21] calculates the memory address of the system library function named “system()” and the memory address of the string “/bin/sh” that represents the path to a shell program. The attacker carefully crafts a buffer that will exploit a known buffer overflow vulnerability in a system service named lpr. The exploit overwrites the stack in such a way that the string “/bin/sh” is passed to the library function “system().” The end result is that a shell is created. Because the software exploited runs under the root user identifier, zero, the new shell also runs under the root user identifier. Anyone with access to this root shell has full control of the system.

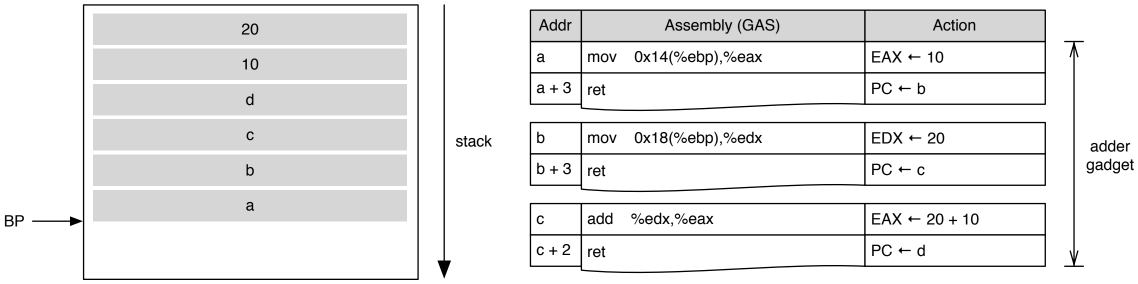
The primary characteristic that distinguishes this kind of attack from the other return-oriented techniques listed later is that it uses a system library function in its entirety to carry out a malicious objective opposed to many smaller pieces of existing authorized system code. Later in 2001, this technique was extended to include the execution of multiple functions [22].

Return-Oriented Programming

In 2005, Krahmer introduced a method that resembles a return-into-libc attack but instead of calling library functions his method borrows “chunks of code” from library functions [2]. He demonstrates how to chain together many chunks of code to carry

[1] CDECL calling convention

out useful tasks. In 2007, Schacham generalized the technique and coined the term *return-oriented programming* (ROP) [3]. Schacham refers to a combination of code chunks as a *gadget*. Like the return-into-libc attack, a return-oriented programming attack is based on stack manipulation. However, rather than attempting to change a single address on the stack, for example the return address as in previous examples, the attacker attempts to place multiple address and other values on the stack.



Addr	Assembly (GAS)	Action
a	mov 0x14(%ebp),%eax	EAX ← 10
a + 3	ret	PC ← b
b	mov 0x18(%ebp),%edx	EDX ← 20
b + 3	ret	PC ← c
c	add %edx,%eax	EAX ← 20 + 10
c + 2	ret	PC ← d

Figure 2.1. Return-oriented adder gadget

Unlike the return-into-libc attack, the addresses are not location of functions in memory, rather locations of segments of code that perform some useful computation and occur just before a return instruction. After the computation is complete another return instruction is encountered that causes the next return address to be popped off the stack and execution continues to the next useful chunk of code. Figure 2.1 illustrates an “adder gadget.” The purpose of this gadget is to add two arbitrary integers that are stored in the stack area of memory. The stack is represented on the left and it has already been populated with values strategically by the attacker. The variables a through d represent the starting addresses of four distinct non-contiguous chunks of code. The resulting sum is stored in a register that could be then used by successive gadgets.

The computations may pop values from the stack or use values in registers. Therefore, along with return addresses, the stack is filled strategically with values to be used in the computations. For example, if the top of the stack holds the address of an instruction that pops the value off the stack into a register and the next value on the stack is an integer value, then when the return instruction is reached the register has the value from the stack that can be used for the next computation. Many computation-return combinations are strung together to create “gadgets.” Shacham shows that enough gadgets can be assembled out of the standard C library to make the technique turing-complete on the x86 architecture.

Jump-Oriented Programming

Jump oriented programming (JOR) is similar to to ROP. It reuses chunks of code contained in libraries to carry out an attack. However, rather than reusing chunks of code that end with return statements, JOP reuses chunks of code that end with jump statements [23]. In ROP, the return statement pops the target address off the stack. In JOP, the jump statement obtains its target address from a register.

At the JOP gadget’s jump target address is a carefully selected chunk of code that represents the “dispatcher gadget.” Only one dispatcher gadget is needed and it is reused as the glue that chains together the other JOP gadgets. The dispatcher gadget has the following two characteristics: First it has a statement that modifies a value in some “predictable” and “evolving” way. A simple example of such modification would be an increment operation. Then the value that has been modified is the target of a jump.

The dispatcher gadget iterates over a dispatch table. In the dispatch table is a list of JOP gadget addresses. The dispatch table does not have to be in executable memory, it can be stored anywhere in memory. The dispatcher table must store values in a way that complements the dispatcher gadget. For example, if the dispatcher gadget increments its value by four bytes, then the dispatcher table must store JOP gadget starting addresses at four byte intervals. As a result, the dispatch gadget becomes a kind of program counter for stringing together JOP gadgets.

Bletsch et. al. show that jump-oriented programming is just as expressive as return-oriented programming. JOP however, is more complex and more difficult to construct. However, it has the benefits of not relying on the stack and is able to evade many ROP detection and prevention mechanisms.

2.2 Mitigation Techniques

Many of the vulnerabilities that lead to exploitation can be remedied by the software programmer. In the absence of bug-free code however, system developers have developed several techniques to prevent, detect, or resist some of the attacks previously described. In this section we discuss such techniques.

2.2.1 Hardware Fault Isolation

In 1968, Dijkstra described the hierarchical division of a computing system into six levels of responsibilities [24]. Software executing in level 0 of the system hierarchy manages the scheduling of tasks and provides a processor abstraction to software executing at higher-level layers. Software executing in levels 1-3 of the system hierarchy manages other computing resources such as memory and input/output devices. The operator, referred to in the model as responsibility level 5, executes programs in level 4. Because software executing in level 0 controls access to the processor, it has a higher *privilege* level than software running in level 1; similarly, software running in level 1 has a higher privilege level than software running in level 2, and so on.

Later, in 1972, Schroeder and Saltzer introduced the design of hardware-based "privilege rings" to support privilege separation for the MULTICS operating system [25]. Similar to the hierarchy suggested by Dijkstra, the ring with the highest privilege level is referred to as "ring 0" and higher level rings have decreasing privileges. Rings with more privileges include all of the capabilities of and access to the memory of the lower privilege rings. Access from a lower privileged ring upward is mediated by a well defined set of functions. These functions are generally referred to as "system calls." This separation of privilege is referred to as hardware fault isolation (HFI). The faults that occur in lower privilege rings are isolated, by the hardware, from disturbing software executing at a higher privilege level. This separation is a course-grained realization of the principle of “least privilege” that states “every user of the system should operate using the least set of privileges necessary to complete the job” and “limits the damage that can result from an accident or error” [26].

Many modern computer processors, such as those found in personal computers, include support for privilege rings or modes [12, 13]. However, many modern commodity operating systems, including Apple Mac OS X (OSX), Microsoft Windows (Windows), and Linux, only use two processor privilege levels: one privilege level for user tasks and one for for supervisor tasks. The modes of processor operation are referred to as “user mode” and “kernel mode” respectively. Tasks running in kernel mode are able to execute all instructions including *privileged* instructions. In contrast, tasks running in user mode are restricted to running a subset of instructions that excludes privileged instructions. The transition between user and kernel mode, via a system call, requires a system interrupt and is therefore computationally more expensive than executing without a mode switch [27].

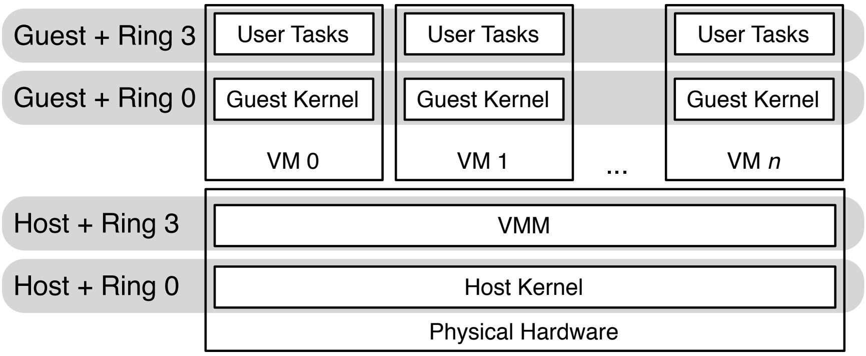


Figure 2.2. Privilege execution modes

Some modern processors also include support for an additional more privileged mode that is used for hardware-assisted virtualization [28,29]. The processor supports a “host” and “guest” mode. When hardware-assisted virtualization is employed, the virtual machine monitor (VMM) and host operating system runs in host mode and each virtual machine (VM) runs in guest mode. Figure 2.2 illustrates privilege rings combined with these virtualization modes. Virtualization-based security mechanisms such as NICKLE and SecVisor take advantage of this additional layer of privilege by placing protection mechanisms in the host that are designed to protect the guest operating systems [18,30].

2.2.2 Reference Monitors

In the "Anderson Report," published in 1972, Anderson et. al. described the "reference monitor" concept [31]. A reference validation mechanism (RFM), that realizes the reference monitor concept, mediates the *operations* performed by a system *subject* on a system *object*; read and write are two examples of an operation users and processes are two examples of a subject; files and I/O devices are two examples of an object. The reference monitor uses an access matrix to determine the access rights between subjects and objects.

Any executing program that is not part of the security mechanism must access system objects through the reference monitor. Anderson sets out the following three requirements for such a mechanism: (1) it must be tamperproof, (2) it must always be invoked, and (3) it must be a small enough program that its correctness can be verified.

Many security mechanisms can be loosely classified as reference monitors because they attempt to enforce memory access control between subjects and objects. However, the requirements are not easily satisfied. For example, any in-kernel security mechanisms share vulnerabilities with the kernel and are therefore not *tamperproof* unless the kernel itself is tamperproof. Additionally, when integrated into a general-purpose operating system kernel, the trusted computing base (TCB) is large and not easily verified. Some VMM-based security mechanisms approximate a RFM [18,30,32,33] because the mechanism is placed outside of the protected system and therefore mostly isolated from the vulnerable system. Additionally, because a

VMM has a specialized set of responsibilities, it can be relatively lightweight and easier to prove correct than a general-purpose operating system [32].

2.2.3 Control Flow Integrity

A software vulnerability is defined, by Bishop and Daily as “an authorized state from which an unauthorized state can be reached using authorized state transitions” [34]. The state transitions are authorized. However, the sequence of state transitions is not authorized. If a security mechanism calculates ahead of time all valid execution sequences and can verify control flow integrity (CFI) during run-time, then it may be able to dynamically detect and prevent the system from reaching unauthorized states.

Some of the other mitigation techniques described can be weakly referred to as CFI verification and enforcement such as the stack canaries technique described in Section 2.2.5. The stack canaries technique prevents the system from following an unauthorized control flow sequence. However, Adabi et. al. describe a stronger CFI design [35]. Ahead of time a control-flow graph (CFG) is built that includes all of the valid control flow paths. Then the CFG is checked during run-time to verify CFI.

Performing CFI verification after every instruction fetch would be extremely computationally expensive. Adabi et. al. perform the verification step at the function level. They instrument the code with two new custom call and return instructions. The new instructions operate on unique identifiers, rather than memory addresses. Then the transitions from one function to another can be compared to the CFG. They show experimentally that CFI enforcement incurs a 45% overhead in the worst case.

2.2.4 NX

One general approach to prevent the introduction of code by an attacker is to make regions of memory that are designated for data storage non-executable. Historically this was done by leveraging memory segmentation as in the non-executable stack technique described in the following subsection.

In addition to memory segmentation, many modern processor architectures support a page-level permission option for restricting memory execution at the hardware-level [12, 13]. This memory page option is often referred to as the “NX bit.” If the NX bit is set, the processor is not permitted to perform an instruction fetch from that memory page.

Following are two techniques that enforce non-executable protection schemes.

$W \oplus X$

A program residing in memory has multiple distinct sections. Some of the sections contain data and some of the sections contain code. Generally a section contains either code or data but not both. A section may hold both code and data as described in Section 2.2.4 but this is atypical. As a result, sections that contain data rarely need to be executable.

Once machine code has been loaded into memory, it rarely changes. Under certain circumstance, such as run-time kernel patching, code may be modified after it has been loaded [36]. However, this is atypical. As a result, sections that contain code rarely need to be writeable.

Some operating systems therefore often enforce a $W \oplus X$ memory access control policy. This policy ensures that no area of memory can be both writeable and executable. At load time, memory sections that contain code are set to read-only and memory sections that contain data are set to non-executable.

An attacker who wishes to add his own executable payload to process memory will fail if the system is enforcing a $W \oplus X$ memory access control policy. If he adds code to a data section, the malicious code will not be executable. If he attempts to add code to a code section, the write will fail because the section is read-only.

A $W \oplus X$ access control policy does not protect systems from ROP or JOP attacks because those attack techniques do not rely on new code being added to the system. Additionally, this technique has a couple of NX bit-related implementation limitations. First, sections of memory may not be aligned on page boundaries. As a result, some memory pages may contain both code and data. These pages are sometimes referred to as "mixed pages." Mixed pages cannot be protected by a page-level access control mechanism such as the NX bit without assistance from other security mechanisms such as those presented in hvmHarvard [33].

Second, in-kernel page-level hardware-based access control mechanisms can be switched off by an attacker with supervisor-level permissions. Therefore, any hijacked process with sufficient privileges can disable the protection scheme. For this reason, virtual machine based security systems such as NICKLE, hvmHarvard and SecVisor have been created to enforce the $W \oplus X$ protection scheme from outside of the protected system [18, 30, 33].

Non-Executable Stack

In the case of a stack-based buffer overflow, the attacker already controls part of the stack memory. At the same time that the attacker is manipulating the return address by smashing the stack, he could write an instruction sequence into the stack memory region and set the return address to the beginning of that sequence. This technique was used by the first Internet worm described by Spafford in 1989 [20].

In 1997, the a Linux kernel developer introduced a set of Linux kernel patches that prevented the execution of code located in the stack memory region [37]. This approach prevented attackers from introducing executable code while smashing the stack. However, it does not protect the return address from being modified. The same kernel developer that introduced the non-executable stack simultaneously introduced the return-into-libc attack previously mentioned [21].

This technique is not compatible with some existing programming techniques. For example, in GCC nested functions are permitted. A nested function is a function that is declared inside of another function. The compiled code that represents a nested function is very similar to a normal function. However, the nested function shares the stack of the parent function. As a result if one wants to use a pointer to a nested function, some care must be taken to first set the stack pointer. GCC implements this using trampolines. A trampoline is a small portion of code that it is written to the stack and then executed. The trampoline sets up the stack pointer appropriately and then jumps to the nested function. The trampoline will fail if the stack is set to non-executable. To accomodate this situation and other similar situations, the system must occasionally relax the non-executable stack restriction.

2.2.5 Stack Canaries

Attackers often use stack-based buffer overflow exploits to hijack program control-flow. Non-executable memory protection techniques do not prevent stack-based buffer overflows, they merely prevent the attacker from executing his own code after he has already hijacked the control flow.

The system can prevent stack-based buffer overflow control flow hijacking if it can detect that a buffer overflow event has occurred. To detect that a stack-based buffer overflow has occurred, some systems implement “stack canaries” [38, 39]. A stack canary is a special value stored on the stack by the operating system upon function entry. The operating system preserves a copy of the special value for verification purposes. The special value is dynamic and is not easily obtained by an attacker. Before the function pops the return value off the stack, the operating system verifies that the stack canary value matches the stored canary value. If the stack has been smashed, then the canary will be overwritten and verification will fail.

2.2.6 System Randomization

Some attacks, such as the return-oriented techniques previously described, require the memory addresses of specific kernel instructions. An attacker must be able to predict the correct addresses or the attack will fail and the exploited program will likely encounter a fault. Depending on the exploited vulnerability, the attacker may be able to guess the addresses at run-time. However, often these attacks are crafted for predetermined memory layouts [40]. Techniques have been developed to make the addresses of such components unpredictable using various randomization techniques such as *address space layout randomization* (ASLR), Instruction Set Randomization (ISR), and Data Structure Layout Randomization (DSLR) [41]. We introduce a new protection technique in Chapter 4 that fits into this category.

Address Space Layout Randomization

When a commodity operating system loader prepares a program for execution it lays out various sections in memory including the program text, user stack, user heap, and memory mapped files including dynamic system libraries. If ASLR is not employed, the location of these key sections is predictable. If ASLR is employed, then the location of these key sections is unpredictable because the starting address of each section is different upon every new execution of the program[2].

Shacham et al. demonstrated that ASLR on 32 bit x86 architectures is vulnerable to brute force attacks [40]. The primary reason for this vulnerability is that only 16 bits of a 32 bit address are randomizable as the result of architecture limitations such as memory alignment. If the attack is repeatable therefore, a brute force attack takes a relatively short amount of time to succeed. The authors recommend moving to a 64 bit architecture to remedy this vulnerability.

[2] some systems, such as PaX on Linux, do not rerandomize the layout of child processes

Instruction Set Randomization

Kc et al. introduced another randomization technique named Instruction Set Randomization [42]. This technique randomizes the machine instructions of a loaded program. When an instruction is fetched from memory it must first be decoded prior to running on the processor. Malicious code that is injected into the loaded program memory would not be randomized in the same way and therefore fail to run as intended.

Some specialized processors have support for similar run-time transformations. However, without hardware support, the technique has significant performance costs.

Data Structure Layout Randomization

Data Structure Layout Randomization (DSLR) is a protection technique that randomizes the layout of data structures at compile-time. Attackers often construct attack code for specific data structure layouts. For example, it is common for a kernel rootkit to hide itself from the process list to evade detection. To do this the rootkit must manipulate the list of tasks. In the Linux kernel, these tasks are represented by the “task_struct.” ADSLR can be employed to randomize the layout of task_struct and prevent this rootkit behavior.

Our randomization work, as described in Chapter 4, is partially inspired by previous work in DSLR by Lin et. al [43]. Our record field order randomization technique is similar to the data structure layout randomization method described therein.

We build upon their work with two separate but related efforts. First, we introduce a technique for automatically determining the suitability of a record for randomization. This is listed as a limitation of the previous work. Second, we introduce a novel polymorphing software technique.

2.2.7 ROP Defenses

ROP Detection

Chen et. al. introduced one of the first techniques for detecting ROP attacks [44]. Their system is named DROP. To detect an ROP attack, DROP dynamically intercepts all return instructions. If the corresponding return address popped from the stack is within the memory region occupied by libc, then DROP begins counting instructions until another return instruction is encountered. If the number of instruction is less than some predetermined threshold, then they classify that sequence of code a potential gadget. If at least three potential gadgets are detected in sequence and the total number of sequential potential gadgets is greater than a second predetermined threshold, then the DROP reports the presence of an ROP attack. This method uses dynamic binary instrumentation and is therefore computationally expensive averaging slow down factor of about five. Though computationally inefficient, the detection rate for the tested attacks was strong with a low false-positive rate.

ROP Prevention

Li et. al. observed that ROP attacks could be prevented by instrumenting the call and return instructions to include one level of indirection [45]. When a call is made, the return address is stored in a centralized lookup table rather than on the stack directly. The table index of the return address stored on the stack. When the function returns, the index is popped off the stack and used as a lookup key to calculate the return address. The program then jumps to the return address. Program flow never jumps directly to the return address stored on the stack. If the attacker were to smash the stack and overwrite the return address index, the table lookup would fail or control flow would jump to an instruction immediately following a previous call site. This technique does not prevent JOP attacks described in [46].

2.3 Kernel Specialization and Minimization

2.3.1 Kernel Specialization

Operating systems kernels often provide a facility for adding code to or activating code in the kernel during compile-time, boot-time, or at run-time. Every operating system implements such mechanisms in a unique way however, the designs are similar. In Linux and BSD, for example, a user may choose from an extensive list of features to be included in the kernel at compile-time by selecting from a list of configuration options. At boot-time, some operating systems include the ability to pass boot flags to the kernel. The flags instruct the kernel to activate or deactivate the corresponding code (and data). At run-time, kernels may often be augmented by loadable kernel modules (LKMs) [10, 47]. LKMs are often used to introduce hardware-specific code into the kernel such as device drivers.

Kernels that support LKMs are "dynamically extensible." They allow for *additive* run-time specialization but the original kernel image remains unmolested. However, some modern kernels contain self-patching kernel code; they may overwrite executable kenel instructions in memory *after* load-time. Such dynamic patching may occur for a variety of reason including: CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging [36].

Pu and Massalin took specialization further in *The Synthesis Kernel* [48] (Syntehsis). Synthesis contains a *code synthesizer* that performs run-time kernel routine specialization for performance enhancements. The techniques employed by the code synthesizer resemble a few common compiler optimization techniques including *constant folding* and *function inlining.* In Synthesis, frequently visited routines are optimized for specific run-time conditions. If the specific conditions are met, then the code branches from the unoptimized routine into the specialized routine.

2.3.2 Operating System Minimization

The hardware rings described in Section 2.2.1 are used by many operating systems to separate user-level execution from kernel-level execution. However, it is not always clear what code should be included in the kernel space. This topic has been hotly debated by kernel designers [49].

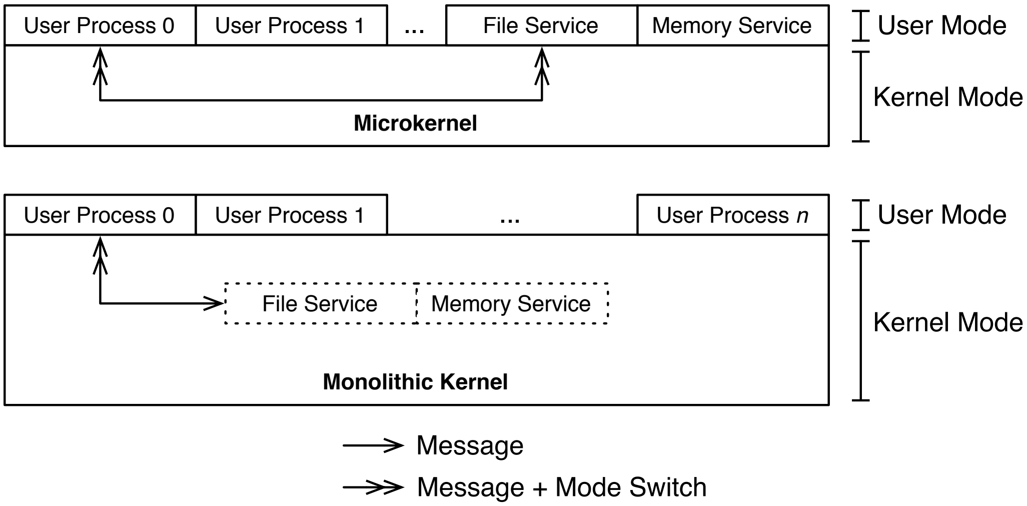


Figure 2.3. Microkernel and Monolithic Kernel Designs

The objective of a *microkernel* kernel design is to minimize the amount of code executed in kernel mode. In this type of kernel, as many kernel features as possible are moved into user space [50, 51]. The kernel then behaves more like a server in a client-server construction. Some potential advantages of this approach are fidelity and manageability. A potential drawback of this approach is increased mode switches that, as previously discussed, are computationally expensive.

The objective of a *monolithic* kernel design is to maximize efficiency. In this type of kernel, many kernel features are included in kernel space that do not strictly need kernel-level privileges. Some potential advantages of this approach are performance and ease of development. Some potential drawbacks of this approach are suboptimal fault isolation and software management.

In 1975 Saltzer and M. Schroeder, in their seminal work titled *The Protection of Information in Computer Systems*, described the security principle: "economy of mechanism." The economy of mechanism states that all system components, security components in particular, should be "as simple and small as possible" [26]. A microkernel design follows more faithfully this principle. In point of fact, a general-purpose microkernel named seL4 has been created and completely formally verified [52]. In contrast, Andrew Tanenbaum, a proponent of microkernel design, refers to monolithic design as "the big mess" because of its perceived lack of structure and ad-hoc design.

Despite its advantages, microkernels have not gained popularity. All of the commodity, general-purpose operating systems available today have a monolithic kernel design. Two notable operating systems with microkernel designs, GNU Hurd and Tanenbaum's MINIX, have had significant technical barriers that ultimately hindered their development and acceptance [49, 53].

In 2003 Bryant et. al. observed that general-purpose operating system kernels often have system calls and kernel subsystems included that are not essential to support the applications of the system [54]. For their Poly^2 framework they use static analysis of the applications of the system and aggressively remove non-essential code from the kernel at compile time. As a result, each kernel in a Poly^2 network is specialized for a specific set of applications. The code removed from the kernel is no longer exploitable.

Where Poly^2 aims to minimize a commodity, general purpose operating system, specialized operating systems, like Choices, also exist [55]. In Choices, like Poly^2, the operating system is tailor-made for a specific set of applications. However, the operating system itself is designed in a modular way and the selection of components is a natural part of the system construction process. Whereas Poly^2 removes components that already exist.

3 VALIDATING THE INTEGRITY OF INCLUDED KERNEL COMPONENTS

Portions of the work described in this chapter are published in the proceedings of the 2012 Military Communications Conference (MILCOM) [36].

3.1 Introduction

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

Attackers can exploit vulnerable programs that are running with elevated permissions to insert kernel rootkits into a system. Security mechanisms similar to NICKLE have been created to prevent kernel rootkits by relocating the vulnerable physical system to a guest virtual machine and enforcing a $W \oplus KX$ memory access control policy from the host virtual machine monitor (VMM) [18]. The $W \oplus KX$ memory access control policy guarantees that no region of guest memory is both writable *and* kernel-executable.

The guest system must have a way to bypass the $W \oplus KX$ restriction to load valid kernel code, such as kernel drivers, into memory. To distinguish between valid kernel code and malicious kernel rootkit code, NICKLE [18] and others [30] [33] use cryptographic hashes for code validation. Offline, a cryptographic hash is calculated for each piece of valid code that may get loaded into the guest kernel. Online, the

VMM intercepts each guest attempt to load new kernel code and calculates a hash for the code. If the online hash matches an offline hash, the load is allowed.

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

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

Our system is guest-transparent; no modifications to the guest operating system are required. However, the whitelist construction is dependent on the guest kernel. Each guest kernel may patch itself in different ways. As we describe in Section 3.3, the whitelist creation procedure requires knowledge of the guest kernel to collect all of the possible valid patches. We discovered that the Linux kernel has six different facilities that influence code modification. We describe each facility and its impact on whitelist creation in Section 3.4.

The implementation of our instruction authentication procedure, described in section 3.4, is part of our reimplementation of NICKLE [18]. NICKLE is a guest-transparent virtual machine monitor (VMM) that prevents unauthorized code from executing with kernel-level privileges. NICKLE, as originally implemented, does not take advantage of the hardware-assisted virtualization extension present in many current processors. Our implementation, named NICKLE-KVM, takes advantage of these extensions by adding NICKLE functionality to the Linux Kernel-based Virtual Machine (KVM) virtualization infrastructure.

Additionally, the original NICKLE does not support self-patching guest kernels. NICKLE-KVM, with our instruction authentication subsystem, removes this restriction. Two other systems similar in nature to NICKLE, SecVisor [30] and hvmHarvard [33], also require that the guest does not contain self-patching kernel code.

3.2 Chapter Organization

The remaining content of this chapter is structured as follows: In Section 3.3, we outline the problem and describe the design of our solution. Section 3.3.1 describes the context of the problem and provides a motivating example. In Section 3.3.2 we describe our solution in detail and demonstrate how it provides a remedy to our motivating example. Section 3.4 describes the implementation of our system and we report our experimental results in Section 3.5. Finally, in Section 3.7, we recap our findings and suggest some future work.

3.3 Design

3.3.1 Problem

Preventing kernel rootkit injection is a challenge. Software is often buggy. If a bug is exploited in kernel mode, then malicious software can gain a foothold into the system with maximum privilege. Once such privilege is attained, the attacker can effectively disable all other protections. For example, similar to NICKLE, some operating systems implement a $W \oplus X$ memory access control scheme for kernel code. Unfortunately, a $W \oplus X$ policy cannot be reliably enforced by the same system that it is trying to protect. Because the operating system has full control over the hardware, a bug may allow an attacker to disable all protections. Systems similar to NICKLE solve this problem by relocating the vulnerable physical system to a *guest* virtual machine (VM) and providing $W \oplus KX$ access control policy enforcement from the *host* virtual machine monitor (VMM). This relocation enables such systems to provide services *below* the operating system [56]. In particular, the host can strictly control guest access to memory.

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

Code Authentication

Code loaded into kernel space originates in executable files stored on disk. Executable file formats divide executable files into sections. At least one section contains machine instructions. The most primitive executable file contains one primary section that holds the machine instructions. We will refer to this section as the *text section*. In addition to the text section, some executable files have many other sections that contain machine instructions. We discovered 11 such unique sections during our experimentation (see Section 3.4 for details). We refer to all sections of a single executable file that contain machine instructions as *executable texts*.

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

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

Code Modification

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

As a motivating example, Linux kernel versions greater than 2.5.68 include support for "alternative instructions" (altinstructions[1]) [9]. Altinstructions enable the kernel to optimize code at run-time based on the capabilities of the CPU. The example provided in the original kernel mailing list announcement was for "run-time memory barrier patching" (RTMBP). According to the RTMBP announcement the default Linux memory barrier instruction sequence, `lock; addl $0,0(%%esp)`, is slow on Pentium 4 processors and should be replaced with the processor's built-in memory barrier instruction: `lfence`. To take advantage of the CPU's capabilities, the Linux distributor could package a separate kernel image tailor-made for the Pentium 4

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

processor or dynamically patch the instructions at run-time using the altinstructions kernel feature.

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

3.3.2 Approach

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

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

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

Patch Set Creation

Patch set creation varies widely and is dependent on the self-patching mechanisms present in the guest kernel. During our experimentation with a Linux guest kernel, we discovered six unique facilities with which the kernel patches itself. We list them in Section 3.4 and describe some of the challenges they pose to patch-level validation. For now, we reintroduce our motivating example of RTMBP to demonstrate the creation of our patch set.

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

Our patch set creation procedure adds two patch definitions for each alt_instr. The first patch definition corresponds to the default instruction, such as "`lock;`

`addl $0,0(%%esp)`" for RTMBP, located in the text. The second patch definition corresponds to the replacement instruction, `lfence` for RTMBP, described by the alt_instr. Both definitions have the same patch-location values and the same patch-length values because they are both candidates for the same patch site. When patch-level validation occurs at run-time, the instruction present in memory must match one of the candidates.

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

Run-Time Patch-Level Validation

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

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

If patch-level validation succeeds, then the write is permitted. We temporarily remove the read-only restriction allowing the guest to execute one write instruction, the one that triggered the write fault, and then we reapply the read-only access restriction. This procedure is illustrated in Figure 3.1.

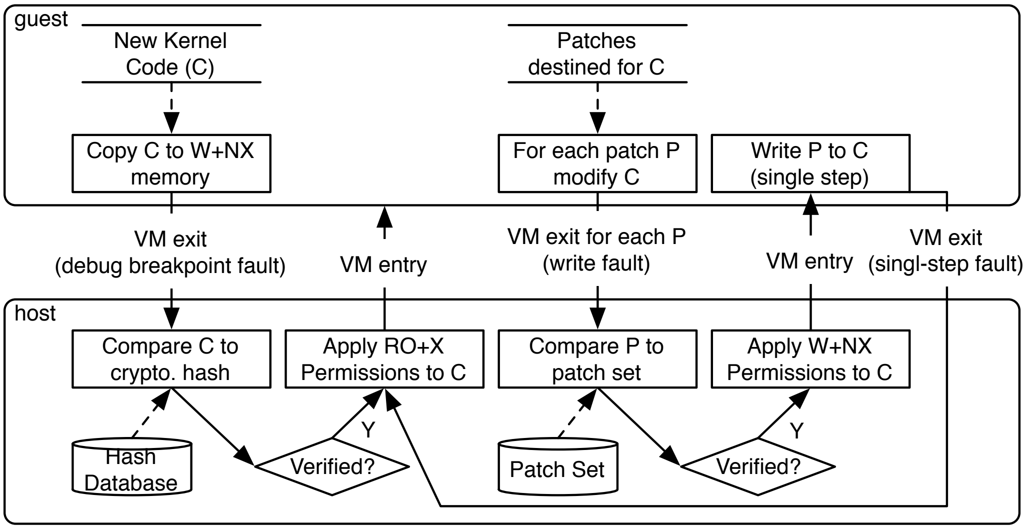

guest
New Kernel Code (C)
Patches destined for C
Copy C to W+NX memory
For each patch P modify C
Write P to C (single step)
VM exit (debug breakpoint fault)
VM entry
VM exit for each P (write fault)
VM entry
VM exit (singl-step fault)
host
Compare C to crypto. hash
Apply RO+X Permissions to C
Compare P to patch set
Apply W+NX Permissions to C
Hash Database
Verified?
Y
Patch Set
Verified?
Y

Figure 3.1. Run-Time Patch-Level Validation

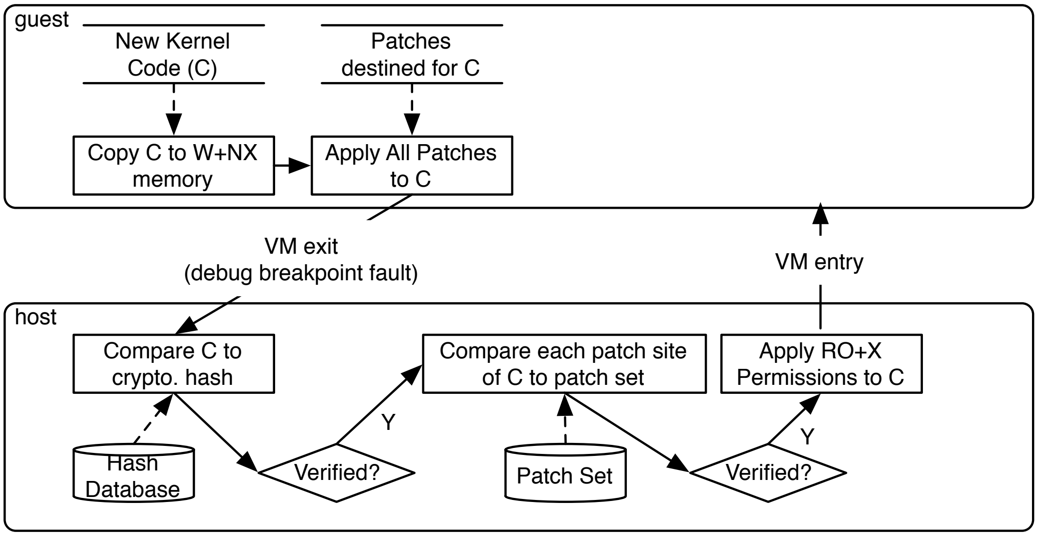


Figure 3.2. Load-Time Optimized Patch-Level Validation

Load-Time Optimized Patch-Level Validation

For our system, each write-fault causes two VM exits. The first is the result of the write-fault itself, and is unavoidable for systems similar to NICKLE in the presence of a self-patching kernel. The second VM exit allows us to reapply the read-only access control restriction to kernel code. VM exits are expensive, therefore we offer the following optimization that significantly reduces the number of VM exits caused by self-patching kernels.

During experimentation we discovered that many changes made by self-patching kernels happen immediately after the code is loaded into memory and before it is executed. Recall from Section 3.3.1 that if we try to compare the cryptographic hash of the code *after* it has been modified, then the hashes will not match. However, we can adjust the hash generation procedure to accommodate self modifying code. Recall from Section 3.3.1 how NICKLE calculates a cryptographic hash in the presence of relocatable code; it writes all zeroes to the relocation sites prior to hash calculation. We use a similar method for load-time optimized patch-validation. From the patch set we know all of the patch sites present in the code. We write zeroes to those locations prior to hash creation. When online, we *defer* validation until after the code has been loaded and modified.

At validation time we first check the integrity of the code using the hash database. Secondly we check the integrity of each unique patch site using the corresponding patch set. If the hash matches and the contents of each patch site are validated, then patch-level validation succeeds and the code is added to authenticated memory. Figure 3.2 demonstrates this procedure in the context of NICKLE-KVM. As shown in the figure, the optimized approach incurs no additional VM exits for patch-level validation.

3.4 Implementation

Our patch-level validation procedure is implemented as a subsystem of NICKLE-KVM. NICKLE-KVM is a NICKLE-like system based on KVM. KVM is a Linux-based VMM that takes advantage of hardware-assisted virtualization. The original NICKLE implementation, based on QEMU, does not take advantage of hardware-assisted virtualization. Instead, it uses binary translation to intercept each instruction to preserve the NICKLE guarantee: “No unauthorized code can be executed at the kernel level. [18]” Instead of the instruction-level redirection technique used by NICKLE, we use the page-level redirection technique introduced by hvmHarvard and takes advantage of hardware-assisted virtualization for better performance [33]. In the absence of a complete description of NICKLE-KVM, we describe the parts we leveraged for our implementation next.

To evaluate our design, we implemented the optimized patch-level validation solution described in Section 3.3.2 for both kernel and module loading. To support our solution, we modified NICKLE-KVM in the following ways:

1. At both kernel load-time (boot) and module load-time the guest must pass control to the host for code authentication. Therefore, at guest system start up we set a hardware debug break point on the return addresses of the guest Linux functions named “init_post” and “trim_init_extable” for kernel and module authentication respectively. These addresses are reached immediately after patches have been applied making them an ideal point in the loading sequence to trigger our authentication procedures. KVM gives us access to the guest’s virtual CPU (vcpu) allowing us to set the corresponding debug registers for these break points. Our implementation catches the resulting debug exceptions and then performs code and patch-level validation.

2. If code validation succeeds, NICKLE-KVM must apply the read-only and execute permissions to the new code prior to returning control to the guest. We notify NICKLE-KVM of the new code and it handles the manipulation of and enforcement of the page-level permissions.

3. If code validation fails, the host can take protective measures. In the case of kernel validation failing, the host could force the guest to shutdown. In the case of module validation failing, the host could force the module to fail loading by manipulating guest execution. For example, in one version of our

implementation we forced the return value of the function "module_finalize" to be -1 by manipulating a guest vcpu register (EAX).

3.4.1 Experimental Setup

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

3.4.2 Patch Validation

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

Alternative Instructions

Our motivating example of RTMBP, introduced in Section 3.3, was an instance of altinstructions. Recall that the purpose of altinstructions is to allow the kernel to optimize code at run-time based on the capabilities of the CPU. The ELF file that contains the loadable kernel code has two sections used by kernel patching functions related to altinstructions: .altinstructions and .altinstr_replacement. The details of how we derive a patch definition from these sections can be found in Section 3.3.2. The altinstructions patch sites do not change after the code has been loaded for the first time because the the CPU capabilities do not change during run-time.

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

SMP Locks

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

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

Jump Labels

Jump labels were introduced into the Linux kernel to optimize kernel tracing [57]. Prior to Jump Labels, if a developer wanted to add a trace point to his code he included a conditional statement to evaluate if tracing was enabled. If enabled, the code would jump to a code block that provided tracing information. To avoid the overhead

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

associated with the conditional statement, Jump Labels are used. Essentially a Jump Label site in code contains either a jump instruction (JMP) or a NOP instruction. At module load-time, the site contains a JMP instruction and it is overwritten with a NOP instruction. When tracing is enabled for that Jump Label, the JMP instruction is reintroduced. The result is that most of the time tracing logic incurs no overhead (except the overhead associated with the NOP instruction).

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

For each Jump Label location, two patch definitions are created: (<address from code field>, 5, <NOP of size 5[3]>) and (<address from code field>, 5, JMP <address from label field>). Unlike altinstructions and SMP Locks, Jump Label code sites get changed at load-time (NOP inserted) and when tracing is enabled (JMP inserted). For this implementation we are interested in the changes made at load-time (NOP inserted). For kernel tracing to work, NICKLE-KVM must perform patch-level validation at write-fault time as described in Section 3.3.2.

Mcounts

When Linux kernel code is compiled with the "-pg" flag the compiler automatically adds a call to the mcount procedure at the beginning of each kernel function [58]. This feature enables kernel profiling and tracing. To optimize Mcounts, when code is loaded that contains mcount call sites, the kernel automatically replaces the call

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

instructions with a DS segment override prefixes (0x3E) as in the case of SMP Locks. Later when mcount tracing is enabled, the call instruction will be restored.

Because the mcount symbol is relocatable, the patch-location can be calculated based on relocation information found in the .rel.text section. The start of the patch-location is one byte before the mcount symbol address in the text. For each Mcount patch site, two patch definitions are created: (<byte offset of mcount from .rel.text - 1>, 1, 0x3E) and (<byte offset of mcount from .rel.text - 1>, 1, 0xe8). Similar to Jump Labels, mcount call sites will get changed during run-time when tracing is enabled. For kernel tracing to work, NICKLE-KVM must perform patch-level validation at write-fault time as described in Section 3.3.2.

Other Facilities

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

Similar to the four facilities detailed previously, Paravirtual Instructions have a special section in the ELF file (.parainstructions) [59]. The .parainstructions section has an entry for each location in the text that holds an instruction that needs to be modified if the guest is running in a paravirtualized environment. Our guest is not running in a paravirtualized environment. Therefore, we explicitly turn off this feature in the guest kernel by supplying the “noreplace-paravirt” boot option at boot-time. If paravirtualized instruction were to be used in our implementation, further work would have to be done to generate the corresponding patch set entries.

Kprobes [60] (including jprobes and kretprobes) are distinct from the other facilities in that they do not have a corresponding ELF file section for calculating patch set entries. Kprobes are typically used for debugging and each patch site is defined by the end user at debug time. Therefore, Kprobes could work in the presence of patch-level instruction validation if the end user supplied a list of potential probe patch sites to the offline patch set creation procedure. Because Kprobes are inserted at run-time, they do not apply directly to the optimized implementation that we describe here.

3.5 Evaluation

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

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

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

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

by the guest kernel, then patch-level validation failed and the module was prevented from loading.

Our implementation incurs no additional VM exits for patch-level validation. We merely reuse the single VM exit already required by NICKLE-KVM to perform hash validation. In the presence of patches that are applied well after loading, such as Kprobe patches, the less efficient procedure described in Section 3.3.2 is required.

3.6 Discussion

We assume that other commodity operating system kernels may patch themselves at run-time using a procedure similar to the Linux patching mechanisms described in Section 3.3.1. Our patch whitelist generation technique depends on our ability to accurately predict the location, size and contents of all possible patches ahead of time. Our patching technique also assumes that patches will not exceed the size of the original instruction and that patches are applied in-place. Our techniques rely heavily on our analysis of the Linux kernel specifically, it is possible that other self-patching operating system kernels defy these assumptions.

It is possible, because of write buffering and the single-stepping method that we use for run-time patch validation described in Section 3.3.2, that an attacker could craft a malicious patch that is a combination of bytes from multiple valid patch definitions. We can limit this vulnerability by ensuring that a patch-site is the combination of no more than two valid patches. To do this, we must locate all patch definitions that include the faulting address in the range patch-location to patch-location plus patch-length. If a matching patch definition is discovered, then the range of guest memory from patch-location to faulting address, *instruction prefix*, is compared to the corresponding instruction prefix in the patch definition data. If the instruction prefix matches, then we compare the new data, that caused the write-fault, to the corresponding segment in the patch definition. If it too matches, then the validation succeeds. We do not validate the instruction suffix because the existing instruction has already been validated. This ensures that at any given time the instruction in memory is a combination of no more than two authenticated instructions. This reduces the risks associated with malicious control flow manipulations.

The patch-definition whitelist creation technique described in Section 3.3.2 may incur significant one-time costs for initial analysis. For our evaluation of the Linux kernel, we discovered six patching mechanisms. Each mechanism requires a customized patch-definition procedure.

Our code authentication and a $W \oplus KX$ protection policy do not prevent attacks that reuse authenticated code. As a result, systems protected only by these techniques will be vulnerable to ROP-style attacks. Further discussion of ROP countermeasures is provided in Chapters 4 and 5.

3.7 Summary

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

4 INCREASING THE DIVERSITY OF INCLUDED KERNEL COMPONENTS

Portions of the work described in this chapter are published in the proceedings of the 2013 International Performance Computing and Communications Conference (IPCCC) [61].

4.1 Introduction

Organizations have strong economic incentive to standardize the software that they use across their enterprise [62]. Furthermore, organizations have strong economic incentive to choose market-leading software [63]. Such economic incentives have lead to a homogeneity of operating system software among network-connected hosts. This homogeneity increases the risk of mass exploitation of similarly vulnerable hosts. [64, 65].

The vast majority of hosts on the Internet, including mobile clients, are running one of three major operating system families[1]. Within each operating system family there exists many versions of the operating system code. However, relative to the total number of hosts, the number of unique versions of operating systems is minuscule.

Malicious operating system kernel software, such as the code introduced by a kernel rootkit, is strongly dependent on the organization of the victim operating system. The lack of diversity in operating systems enables attackers to craft a single kernel exploit for a single unique operating system version that has the potential to infect millions of hosts.

One approach to strengthen computer security is *software diversity.* Organizations can choose to add diversity to their software ecosystem by purchasing less popular

[1] http://www.netmarketshare.com/

software that performs the same tasks. However, such a choice is often not economically feasible even if viable alternatives exists [62].

Another approach to software diversification is for the software itself to mutate; where multiple variants of the same version of a piece of software are used. The variants are compatible with each other and *function* the same from the end-user's point of view. However, some underlying component of their *structure* has been changed.

Computer virus authors have long used automatic software diversification to avoid detection. Each time the virus infects a new victim the virus mutates. This mutation prevents signature-based anti-virus software from detecting the virus. This class of malware is often referred to as *polymorphic* computer viruses [66, 67]. In much the same way, but inverted, polymorphic software can be used offensively to withstand malicious software. If the vulnerable components of a piece of software have been changed in an unpredictable manner, then attackers must create many unique variations of their exploit to attack vulnerable systems en masse. If enough variants exists, then mass exploitation is much more difficult to achieve if not impossible.

Some attacks, such as the return-oriented techniques introduced by [21], [2] and [3] require the memory addresses of key system libraries. An attacker must be able to predict the correct addresses or the attack will fail and the exploited program will likely encounter a fault. Depending on the exploited vulnerability, the attacker may be able to guess the addresses at run-time. However, often these attacks are crafted for predetermined memory layouts [40]. If the memory layout can be obfuscated in some way, then the exploitation can be thwarted.

Forrest et. al. make a strong case for software diversity and describe a few possible techniques including: adding or removing nonfunctional code, reordering code, and reordering memory layouts [8]. Our technique builds on the latter. We describe two different ways to mutate an operating system kernel using memory layout reordering to resist kernel-based attacks.

Our randomization techniques occur at compile-time. Some software diversification techniques occur at run-time or load-time. Some, such as instruction set randomization, incur significant run-time overhead [42]. In contrast, our techniques incur no run-time overhead and are therefore suitable for any system including low-powered devices such as mobile phones and embedded devices.

Our randomization techniques are tailored specifically for operating system kernels. The techniques themselves have wider application, however they are practically well suited for an operating system kernel. Because our technique occurs at compile-time all dependent software must be recompiled to be compatible. In the user-space, this could be a significant undertaking; if for example, one wanted to randomize a system library all dependent software would have to be recompiled. However, the kernel compilation is self contained, loadable kernel modules notwithstanding, and its code is executed only through well-defined system calls. Our system does allow for loadable kernel modules to be compiled with compatible randomization during kernel compilation or separately.

Practically, we would not expect our techniques to be adopted by every end-user. A small fraction of computer users would have the technical prowess to compile their own kernel and for many operating systems the kernel source code is not available. Rather, our techniques are better suited for organization-wide application. For example, military organizations often distribute their own version of the Linux kernel. If they employed our technique they could withstand all versions of kernel malware that was not specifically targeted for their organization. Similarly a mobile device manufacturer could randomize their kernels so that it would be invulnerable to the same attacks leveraged against other similar devices.

Our contributions are as follows: We introduce a new method for randomizing the stack layout of function arguments. We refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record. We provide an implementation of our techniques using the plugin architecture offered by GCC. Finally, to evaluate the security benefits of our techniques we randomize multiple Linux kernels using our plugins and attack them using kernel rootkits. We show that by strategically selecting just a few components for randomization, our techniques prevent all tested kernel rootkits.

4.2 Chapter Organization

The remainder of this chapter is organized as follows: we begin in Section 4.3 by laying out the design of our randomization techniques followed by a description of our GCC implementation in Section 4.4. In Section 4.5, we evaluate the security merits of our approaches and the performance of our implementations. Finally, in Section 4.7, we summarize our contributions and findings.

4.3 Design

Our design has three distinct but related parts: *record field order randomization* (RFOR), RFOR *suitability analysis*, and *subroutine argument order randomization* (SAOR). Each part is described in the following three sections.

4.3.1 Record Field Order Randomization

We have designed our field order randomization technique to occur at compile-time. During compilation we randomize the field order of the record *definition*. As a result, all instances of that record type are defined with the new field order in the resulting binary. Each compilation unit may have its own definition for a given record. Therefore, we use a seeded pseudo-random algorithm so that the same field order can be replicated across multiple compilation units. This also allows for modular software, such as loadable kernel modules, to be compiled with compatible record layouts.

Our field order randomization technique takes as input the source code of the software to be randomized, a set of one or more record names, a set of one or more randomization seeds, and a set of one or more padding flags. Each record name may correspond to a unique randomization seed or one seed may apply to all record names. Each record name corresponds to a unique padding flag that indicates whether or not the record receives padding. Our field order randomization technique produces as output a compiled binary. Given different randomization seeds and padding flags, our system may produce distinct binaries. However, our field reordering algorithm is not collision resistant; multiple seeds will produce the same field order. Naturally, records with more fields will have more layout permutations.

To increase the possible number of layout permutations, our system takes as input a padding flag for each record name. When this boolean flag is set to true for a given record name a pseudorandom number of bogus fields are inserted pseudorandomly into the record. The randomization algorithm used to add padding is also seeded with the same seed that determines the layout. We have bound the amount of padding automatically generated so that a record will have no more than two times the number of fields of its original. This upper bound was selected arbitrarily. In practice, some bound will be desired to keep the records from becoming space-inefficient.

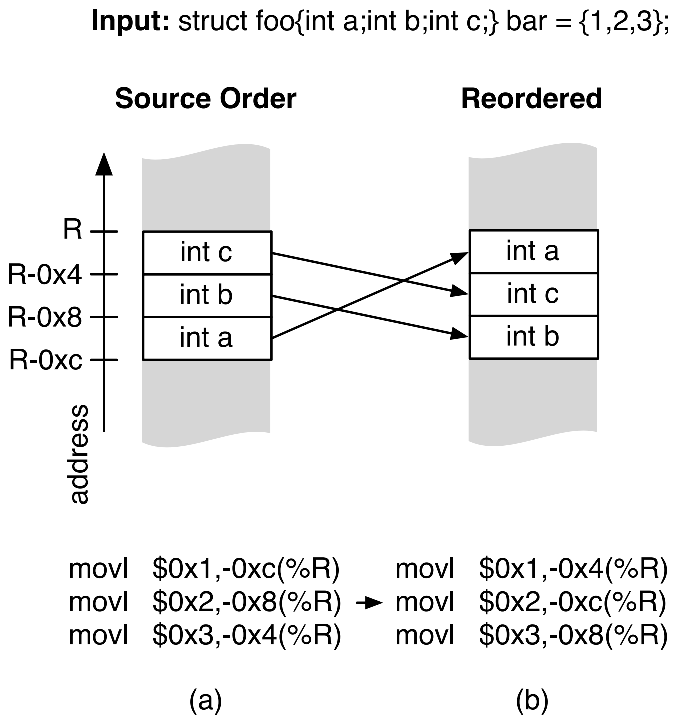
Field order randomization takes place after the source code has been parsed into an abstract syntax tree but before any compiler optimization passes. Figure 4.3, located later in the implementation section (4.4.1), illustrates the abstract syntax tree for both a randomized (b) and unrandomized (c) record. Figure 4.1 illustrates the stack memory layout and machine instructions for assignment to an unrandomized (a) and randomized (b) record.

Given a record type τ, Algorithm 1 depicts field order randomization subroutine. We use the Knuth shuffling algorithm for reordering the fields [68].

When a kernel data structure is reordered using RFOR, attack code compiled, without randomization, against the kernel source code will be incompatible. For example, regarding the record randomization illustrated in Figure4.1, suppose the attack code was trying to assign a malicious value to field "foo.b." The malicious code will assume that "foo.b" is at offset `0x8` and assign the malicious value to offset `0xc` instead. A similar problem occurs when attack code tries to read from a

Algorithm 1 Field order randomization with padding

```
$a[] \leftarrow \tau_{fields}$
$n \leftarrow$ count of items in $a[]$
**if** option to add padding is **true then**
  $x \leftarrow$ seeded pseudorandom integer $< max$ and $> min$
  **for** $1$ **to** $x$ **do**
    $a[] \leftarrow$ new field declaration
    $n \leftarrow n + 1$
  **end for**
**end if**
**for** $i = n - 1$ **down to** $1$ **do**
  $j \leftarrow$ seeded pseudorandom integer $\geq 0$ and $\leq i$
  swap $a[j]$ and $a[i]$
**end for**
$\tau_{fields} \leftarrow a[]$
```



Input: struct foo{int a;int b;int c;} bar = {1,2,3};

```
movl  $0x1,-0xc(%R)      movl  $0x1,-0x4(%R)
movl  $0x2,-0x8(%R)  ->  movl  $0x2,-0xc(%R)
movl  $0x3,-0x4(%R)      movl  $0x3,-0x8(%R)
```

Figure 4.1. Record field order randomization. “R” represents a general-purpose processor register

predetermined offset. The results of the malicious code execution are unpredictable in the presence of RFOR. For our evaluation, detailed in Section 4.5, we found that malicious code often resulted in a non-destructive kernel “oops” (not panic) when a victim data structure was randomized. For all tested kernel rootkits, the malicious software failed to run as intended on systems compiled with RFOR. In some cases, infection was prevented entirely by RFOR.

4.3.2 Suitability of a Record for Field Reordering

Not all records are suitable for field reordering. If a record is used or defined, in a raw memory form, by an external system that expects a predetermined format, then the record may be an unfit candidate for randomization. For example, for our evaluation we randomized the Linux TCP header record (tcphdr). When TCP packets were formed using the randomized layout, the system could not communicate with other systems using TCP. If both end-points had their TCP headers reordered in the same way, then it may be possible for the two to communicate; assuming that deep packet inspection or similar transport-layer logic is not present on intermediary nodes and total header size does not change as the result of field realignment (see discussion of record resizing in Section 4.4.1 and Figure 4.4).

If the compiler supported it, the programmer could annotate the source code to indicate whether or not a record's fields may be safely reordered. For example, some compilers provide a way for the programmer to suggest how to align a variable type. In GCC, the programmer can specify type attributes using the keyword __attribute__ at the end of a type definition. The programmer can specify the "packed" attribute of a record to instruct the compiler not to realign its fields. We can leverage this paradigm for our purposes; we can add a custom attribute to the compiler that indicates that the record is unfit for field reordering. If the attribute is not present, the compiler could freely reorder fields as it sees fit. We describe our implementation of this approach in Section 4.4.1. This approach may have benefits beyond randomization for security purposes. For example, if the compiler were free to reorder fields it may be able to automatically improve record cache performance as described by Chilimbi et al [69].

In the event that such a compiler mechanism were widely adopted today, there exists much source code that is not already annotated. In the absence of compiler support and source code annotations, we have designed a static analysis technique for testing the suitability of a record for field reordering.Our initial design was informed by previous research at Hewlett-Packard in compiler optimization techniques [70]. However, our technique has a different purpose and differs significantly from this prior work.

First we define a few terms: x, y and z are free variables. τ represents the candidate type and $\neg\tau$ represents all other types. A type followed by an $*$ represents a pointer to a variable of that type. τ_n represents field n in τ. The function A returns the memory address of its argument.

There are four conditions that may disqualify a record for field reordering[2]:

$x_{\tau *} = A(y_{\neg \tau})$

If a variable of type τ pointer is positioned on the lefthand-side of an assignment[3] and the righthand-side expression does not result in a τ-pointer, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose that the righthand-side expression is the address of a buffer that is filled by a network receive function and the format of the receive buffer is consistent with the headers of a published network protocol. The protocol specification is external to our system and not subject to compile-time reordering. Therefore, if τ is reordered, then τ can no longer be used to reliably access parts of the network buffer.

$x_{\neg \tau *} = A(y_{\tau})$

If the righthand-side expression of an assignment[3] results in a τ pointer and the lefthand-side variable is not of type τ-pointer, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose the inverse of the previous example listed in Section 4.3.2. Suppose that the righthand-side expression results in a memory address that is formatted as a reordered version of type τ and the lefthand-side is expecting a pointer to a region of memory that specifies network headers to be sent directly on the network, then the send buffer will be formatted in a way that is inconsistent with the network protocol and network communication will fail.

[2] Our system is designed to protect commodity operating system kernels. As a result, our design is pertinent to the C programming language specifically.

[3] In the source code, an assignment may be manifested in various forms. For example: passing a variable into a function is an assignment.

$x = A(y_{\tau_z})$

If the righthand-side expression of an assignment[3] results in the address of a field inside of a variable of type τ, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose that the x is used in the source code to calculate the address of a sibling field. If the sibling offset address calculation were not aware of the potential for field reordering, then the calculation would likely be incorrect. It is worth noting however, that code containing this kind of sibling calculation would be difficult to maintain considering alignment issues alone. However uncommon, it remains a possibility.

τ is a member of a union or record

If τ is the type of a member included in a union definition, then τ may not be suitable for field reordering.

An example of how this use is potentially unsafe for field reordering is as follows: suppose that a union was constructed with two members τ and $\neg\tau$. Supposed that τ_x and $\neg\tau_y$ were the same offset in the union. If τ_x were relocated at compile time, then $\neg\tau_y$ would no longer point to the same offset. Additionally, suitability analysis would have to be performed on all unions that include τ because unsafe casting of the union may occur.

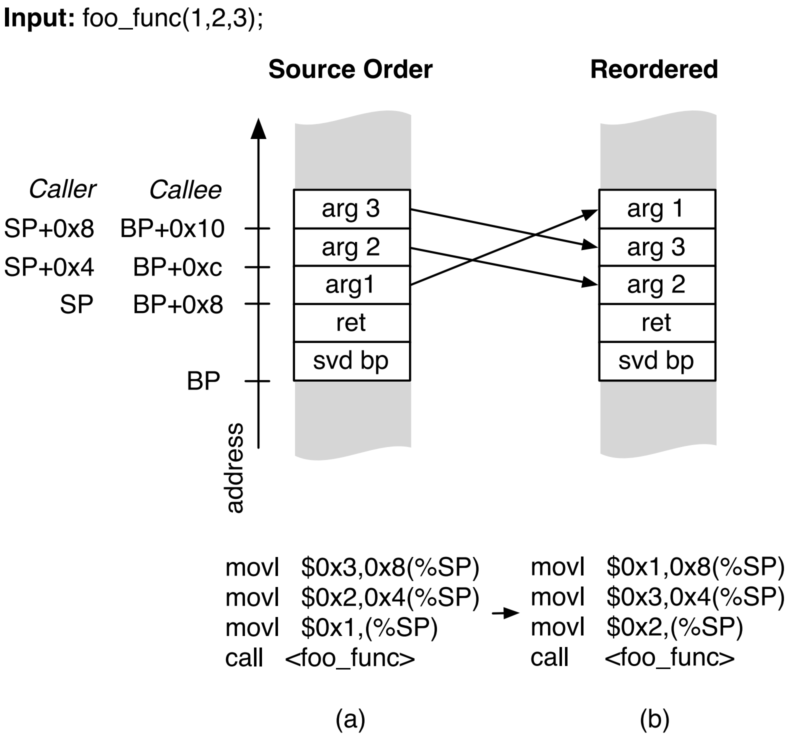
Similarly, if τ is the type of a field included in anther record, then τ many not be suitable for field reordering. Suitability analysis would have to be performed on all records that include τ because unsafe casting may occur.

4.3.3 Subroutine Argument Order Randomization

Similar to RFOR, we have designed our subroutine argument order randomization technique to occur at compile-time. During compilation we randomize the argument

Algorithm 2 Subroutine argument order randomization with padding

```
for all (definitions, types, calls) of S do
  $a[] \leftarrow S_{arguments}$
  $n \leftarrow$ count of items in $a[]$
  if option to add padding is true then
    $x \leftarrow$ seeded pseudorandom integer $< max$ and $> min$
    for 1 to $x$ do
      if S is definition then
        $a[] \leftarrow$ new parameter declaration
      end if
      if S is type then
        $a[] \leftarrow$ new argument type
      end if
      if S is call then
        $a[] \leftarrow$ new call argument
      end if
      $n \leftarrow n + 1$
    end for
  end if
  for $i = n - 1$ down to 1 do
    $j \leftarrow$ seeded pseudorandom integer $\geq 0$ and $\leq i$
    swap $a[j]$ and $a[i]$
  end for
  $S_{arguments} \leftarrow a[]$
end for
```



```
movl   $0x3,0x8(%SP)        movl   $0x1,0x8(%SP)
movl   $0x2,0x4(%SP)   →    movl   $0x3,0x4(%SP)
movl   $0x1,(%SP)           movl   $0x2,(%SP)
call   <foo_func>           call   <foo_func>
```

Figure 4.2. Subroutine argument order randomization. "SP" represents the stack pointer register. "BP" represents the stack base pointer register.

order for each *definition* of, *type* of, and *call* to a given subroutine. Also similar to RFOR, we use a seeded pseudorandomization algorithm so that the same argument order can be replicated across multiple compilation units.

Our argument order randomization technique takes as input the source code of the software to be randomized, a set of one or more subroutine names, a set of one or more randomization seeds, and a set of one or more padding flags. The meaning of each input is analogous to the RFOR example outlined in Section 4.3.1. Our argument order randomization technique produces as output a compiled binary.

When a kernel subroutine is reordered using SAOR, attack code compiled, without randomization, against the kernel source code will be incompatible. For example, regarding the subroutine argument order randomization illustrated in Figure4.2, suppose the attack code made a call to a randomized subroutine. The callee variable that holds the first argument will get the value of the malicious callers third parameter. Similar to RFOR, the results of the malicious code execution are unpredictable.

Not all subroutines are suitable for SAOR. Subroutines that are called using a function pointer are not randomizable because the caller would not be identified at compile-time and the call stack would not be correctly reordered. Additionally, functions with a variable length argument list are likely not randomizable without modifications to the subroutine logic.

For our evaluation, detailed in Section 4.5, we found that Similar to a system using RFOR, malicious software failed to run as intended on systems compiled with SAOR.

4.4 Implementation

To test our design, we implemented three plugins: RFOR, RFOR Fitness, and SAOR using the GNU Compiler Collection (GCC). GCC versions 4.5 and newer have the ability to load user-supplied plug-ins during compilation [71]. We leverage this plug-in architecture to realize our design. Following are the details for each plug-in.

4.4.1 Record Field Order Randomization

The RFOR GCC plugin provides compile-time record offsets randomization. The GCC plug-in architecture provides an event callback named "PLUGIN_FINISH_-TYPE" that gives us a hook into the compilation process. The PLUGIN_FINISH_-TYPE event occurs after a record or union type specifier has been parsed. The event data passed to our plugin's callback function is a pointer to the AST tree node for the most recently parsed record or union as illustrated in Figure 4.3(b) and (c). We are only interested in record types. As a result, we ignore events associated with unions.

We implemented two versions of this plug-in. Both versions perform the randomization step in the same way; using the algorithm described in Algorithm 1. To save the new layout, the pointer fields of each field declaration are updated in the AST as

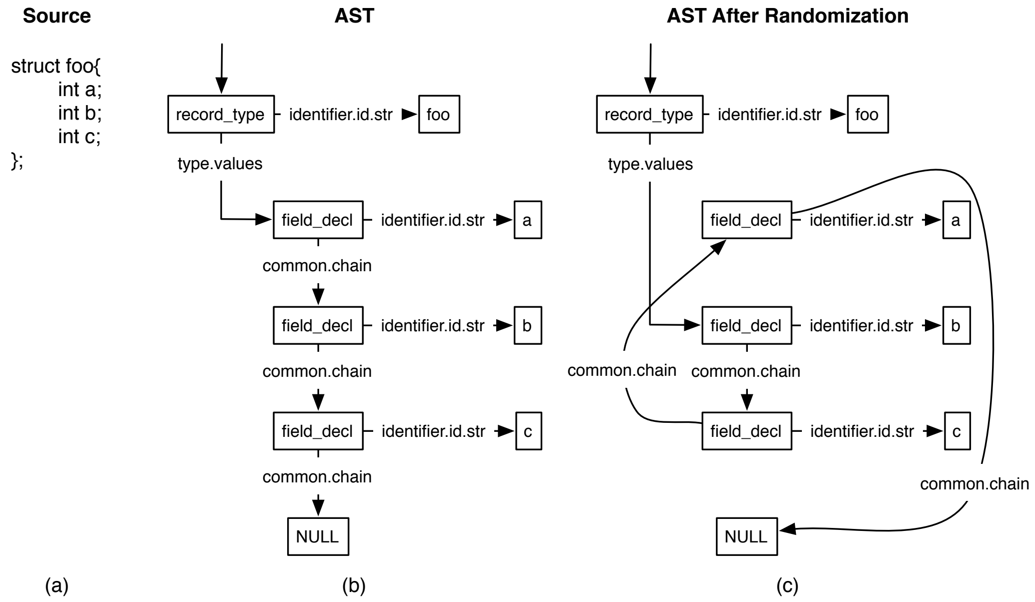


Figure 4.3. AST-based randomization

shown in Figure 4.3(c). The two versions of this plug-in differ only in how the records are selected for randomization. The first version, variant “A” randomizes all records encountered except those with the GCC attribute “noreorder” set in the source code, e.g. “__attribute__ ((noreorder))”.

The second version, variant “B” randomizes only the records provided by name on the command line. The plug-in and its argument are provided by the user as GCC command line flags. The plug-in accepts multiple arguments, one argument for each target record name. All records of the same name will be randomized because the name is not necessarily unique across all compilation units. The randomization happens at type definition time and therefore applies to all instances of the record.

In addition to standard randomization, our plug-in allows the user to specify a boolean flag for each target record that indicates whether or not the record should be padded with bogus members. If yes, our plug-in adds a random number of bogus fields to the target record or records.

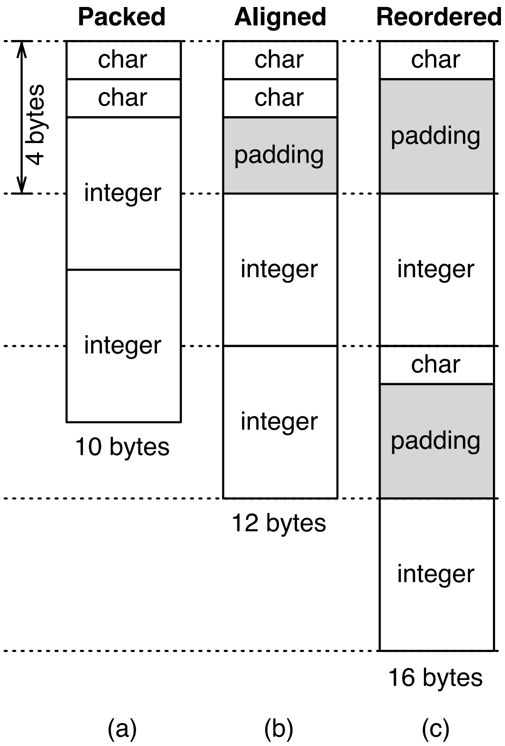
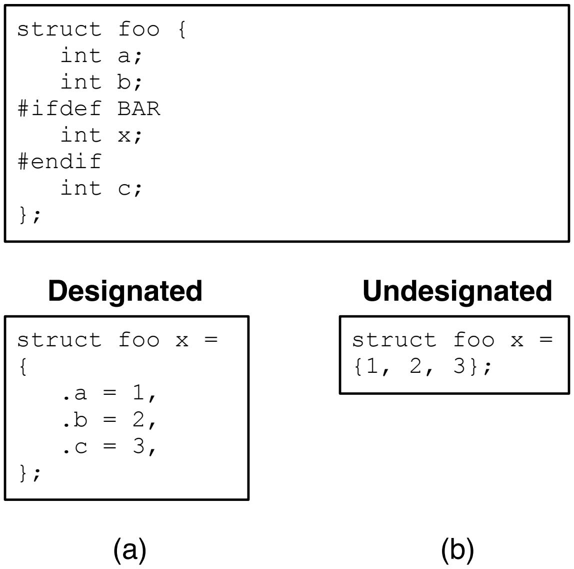


Figure 4.4. Record size variation

The total record size may change as a side effect of field reordering as illustrated in Figure 4.4. Notice that Figure 4.4(c) is a reordering of fields found in (b). If a record is not “packed,” then the compiler may align fields for efficiency. One common optimization is for the compiler to align field offsets on word boundaries as shown in (b) and (c). If a record is packed, as in Figure 4.4(a) the record will always take up the minimal amount of memory and the total record size will not change as a result of randomization.

Our initial implementation performed randomization after the entire compilation unit had been parsed but prior to optimization passes (PLUGIN_PRE_GENERICIZE). We observed that if the size of the record changed as a result of randomization, then some compile-time calculations such as “sizeof” were incorrect. Also affected was offsetof (__builtin_offsetof) calculations that we later discovered was an essential calculation for common Linux kernel data structures such as “list.” GCC, folds the result of sizeof into a constant during parsing and leaves no indication that the constant was a result of the sizeof calculation. As a result, there was no reliable way for us to find the constant in the AST to update it. To remedy this problem, we moved the randomization procedure to the PLUGIN_FINISH_TYPE event as previously described. This event happens after the record definition is parsed but before it is an argument to sizeof and other similar compiler functions.



```
struct foo {
   int a;
   int b;
#ifdef BAR
   int x;
#endif
   int c;
};
```

Designated

```
struct foo x =
{
   .a = 1,
   .b = 2,
   .c = 3,
};
```

(a)

Undesignated

```
struct foo x =
{1, 2, 3};
```

(b)

Figure 4.5. Variable initializers

We discovered that this early randomization approach fixed size-related calculations but broke non-designated variable initializers. Figure 4.5 illustrates the difference between initializers. Currently, this is the only known limitation of our plug-in. The problem is that when a non-designated initializer is parsed, the compiler assumes that the source-code-order of values matches the source-code-order of the record members. This problem could be solved by modifying the way GCC stores initializers in the AST. Simply adding a node attribute that indicates how the initializer was formed would solve the problem. However, we discovered that, for the tested kernel records, non-designated initializers were not used. The reason for this is shown in the Figure 4.5. If the macro BAR is defined, then (b) would be an invalid initializer assuming that the programmer was intending to assign the value 3 to field c and not x. However, with or without defining BAR, initializer (a) would assign values correctly. Our static analysis tool, described in Section 4.4.2, reports when and if the target record is declared with an initializer.

GCC permits an incomplete variable type, such as a flexible array, to be positioned as the last field of a record. Therefore, if the last element is an incomplete type, it cannot be moved during randomization. Our plug-in handles this situation by pinning the last field to the last position when the field type is incomplete.

4.4.2 RFOR Fitness Check

The RFOR Fitness Check GCC plug-in is a static analysis tool for determining the fitness of a candidate record for randomization by inspecting the abstract syntax tree during compilation. The inspection occurs after parsing has been completed but prior to optimization passes.

Again we leverage the GCC plug-in architecture callback event "PLUGIN_PRE_GENERICIZE" to do most of the work. The callback associated with this event received the function definition (FUNCTION_DECL) of the most recently parsed function definition. From this root node we traverse the tree using the API functioned named "walk_tree()." One of the parameters to walk_tree is a callback function name. Our callback function checks for four kinds of nodes: NOP_EXPR, CALL_EXPR, ADDR_EXPR, and CONVERT_EXPR.

A NOP_EXPR node represents, among other things, a cast of one type to another. Figure 4.6 shows the case where a pointer to a record is cast from another type. This case would be reported by our tool. We can use the NOP_EXPR to find the cases where memory is cast to or from a pointer to the type of the target record. If the target record for analysis was of type foo as shown in the figure, then we check to see if the left-hand side of the cast is a pointer to a variable of type foo by checking the corresponding nodes of the AST. If the right-hand side of the modify expression

Input: x = (struct foo *) y;

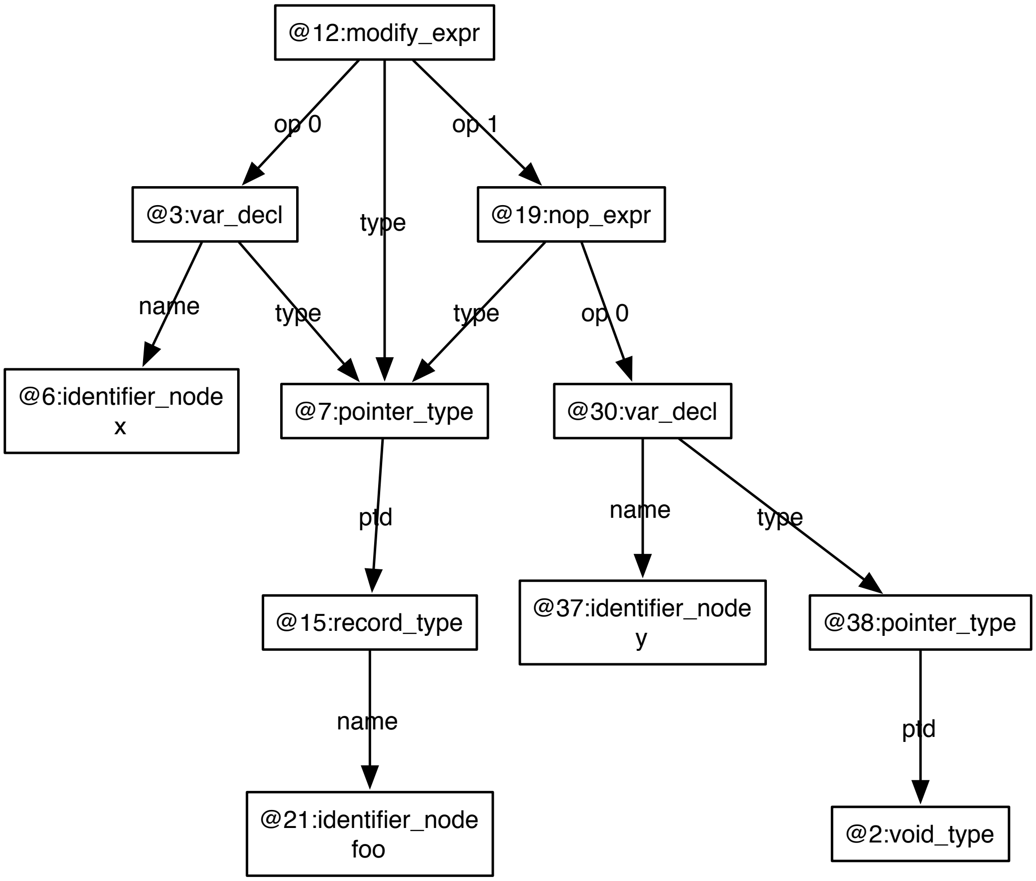


Figure 4.6. AST of a cast from one pointer type to another

is not also a pointer to a variable of type foo, then our plug-in reports this case as a potentially unsafe operation.

GCC does not allow for direct assignment from one record type to another even if both records have the same layout[4]. Similarly, GCC does not allow for coerced assignment from one record type to another using a cast[5]. Therefore, all unsafe assignments to/from a record type will be pointer casts using the NOP_EXPR. Our first two cases: $x_{\neg\tau*} = A(y_\tau)$ and $x = A(y_{\tau_z})$ are therefore tested when a NOP_EXPR is encountered in the AST.

[4] error: incompatible types when assigning to type

[5] error: conversion to non-scalar type requested

The third case: $x = A(y_{\tau_z})$ is tested also during the PLUGIN_PRE_GENERICIZE event. We have developed two detection mechanisms for this case. The first is the more straight-forward and conservative approach. When an ADDR_EXPR is encountered we check the operand. If the operand of the address expression is the field of a target record, then we report the instance. The second implementation of this test is similar, however we only report the instance if the ADDR_EXPR is part of a CONVERT_EXPR and the address is immediately converted to an integer. In this case the the ADDR_EXPR is a child node to the CONVERT_EXPR so we ignore ADDR_EXPR and check the operand of all instances of CONVERT_EXPR. If the type of the convert expression is INTEGER, then we report the instance. More information about the motivation for this optional refinement is described in the evaluation Section 4.5.2.

Often, a record field is passed by address to a function. As described in Section 4.3.2, taking the address of a field and then using it to calculate the address offset of a sibling field is an unlikely occurrence. An even more unlikely occurrence is for such a calculation to be performed on the address of an argument to a function. Therefore, our evaluation ignores cases when a field address is taken in the context of a call parameter. To detect such occurrences, we collect all call parameters during the PLUGIN_PRE_GENERICIZE event that are pointers to fields of the target record type. Then later during AST tree traversal, when the address is taken of the field, the ADDR_EXPR corresponding to call parameters are not reported.

The fourth and final case, the case when the target record is nested inside of a union or record, is tested when the compiler event PLUGIN_FINISH_TYPE is encountered. The event data received is a finished type node. For our purpose we ignore all types except RECORD_TYPE and UNION_TYPE. Each type contains a set of FIELD_DECL nodes. We iterate over each field declaration and check the type. If the target type is found in a FIELD_DECL, the instance is reported as a potentially unsafe condition.

As mentioned in Section 4.4.1, non-designated variable initializers are not compatible with field reordering. This is an implementation limitation. Therefore, as part of our fitness analysis, we report when initializers are used for the target record type. We use the PLUGIN_FINISH_DECL event to check for such cases. Unfortunately, by the time the PLUGIN_FINISH_DECL event occurs the initializers has already been parsed into its final form in the AST. At this point, from the AST perspective, a non-designated initializer is structured the same as a designated initializer. As a result, we report all instances where the target record is initialized. As previously described, a modification to GCC proper could remedy this situation by classifying the initializer nodes as either designated or non-designated in the AST.

4.4.3 Subroutine Argument Order Randomization

The SAOR GCC plugin provides compile-time subroutine argument order randomization. The GCC plug-in architecture provides two callback events that we leverage for this plugin named "PLUGIN_PRE_GENERICIZE" and "PLUGIN_OVERRIDE_GATE." We introduced the former in Section 4.4.1. The event data passed to our plug-in's callback function is a pointer to the AST tree node for the most recently parsed function definition. The latter event, PLUGIN_OVERRIDE_GATE, occurs many times during optimization passes. However, we are only interested in the first occurrence. The first occurrence happens before optimization but after all PLUGIN_PRE_GENERICIZE events.

Randomize Call Expression Arguments

When a PLUGIN_PRE_GENERICIZE event occurs, we traverse the function definition's AST and inspect all call expressions. If the target of the call expression is a function targeted by randomization, then we reorder the call arguments. Because the call expression is allocated with a static number of arguments and to support adding bogus call parameters, we must build a new call expression with the new arguments using the API function named "build_call_array_loc." Once we build the new call expression we replace the original with the new one in the AST.

Also during the PLUGIN_PRE_GENERICIZE event, we capture the unique identifier for the completed function definition if the function is a target for SAOR. This identifier is used later for definition and type randomization. We discovered that we must randomize all of the call expressions prior to randomizing the function definition and type. Otherwise, if bogus fields were added to the definition, then the parser would complain when a corresponding call expression was parsed because the source code would not contain enough parameters; we cannot fix the call expression until after it has been parsed into an AST.

Randomize Function Definition and Type

The PLUGIN_OVERRIDE_GATE occurs after all functions have been parsed into the AST. The difference between this event and PLUGIN_PRE_GENERICIZE is that the latter occurs after each function is parsed. Therefore at the time that the PLUGIN_OVERRIDE_GATE event occurs our AST is in an inconsistent state. All calls to target functions have been randomized however, the function definition and type has not yet been randomized. Recall from the previous section that we had to delay randomizing these components until after all calls have been parsed to avoid compilation errors.

We collected the AST unique identifier for each of the completed target function definitions previously during the PLUGIN_PRE_GENERICIZE event. Now we iterate over that list of targeted function definitions and randomize the definition as well as the associated type for each.

Reordering the arguments for both the definition and the type was similar to RFOR randomization illustrated in Figure 4.3. We obtained parameter declaration head list node and the argument type head list node from the GCC macros DECL_ARGUMENTS and TYPE_ARG_TYPES. Using the algorithm described in

Algorithm 2, we shuffled each list by updating the next pointer of each element. For the type, we used the function GCC build_function_type to reinitialize the type attributes.

4.5 Evaluation

We performed our evaluation on a 2.30GHz four core Intel®Core^{TM}i5-2410M CPU with 8 GB of RAM running 32 bit Fedora (15), Linux (2.6.38.6-26), and GCC (Red Hat 4.6.3-2). Our experiments were performed in KVM/QEMU (0.14) virtual machines with 1024 M of memory allocated for each.

4.5.1 Security Benefits

To evaluate the security benefits of record field order randomization, we employed our RFOR plug-in to compile two different Linux kernel versions, 2.6.26 and 3.3[6], and attempted to run four kernel rootkits against the protected system.

Kernel rootkits in the wild are kernel version-sensitive. As a result, two of the tested rootkits were rewritten to be compatible with our test systems using the principles outlined in the original work. The hidefile rootkit is based on [72]. The hideproc rootkit is based on [73]. The other two rootkits Adore-NG and hp were used in their original form with few compatibility modifications.

One cannot compile Linux kernel versions less than 2.6 with GCC version 4 or greater. The plug-in architecture was not introduced in GCC until version 4.5. As a result, the oldest kernel that we were able to randomize was 2.6.26 using GCC 4.7.3 and our RFOR plugin. We were able to test the adore-ng rootkit against this version of the Linux kernel.

We identified five security sensitive records in the Linux kernel based, in-part, on previous work: task_struct, module, file, proc_dir_entry and inode_operations [74]. We compiled our test kernels using variant B of our RFOR plug-in and specifying,

[6] We used Fedora's custom version 2.6.43.8-1 that is a patched version of vanilla kernel 3.3

depending on the experiment, some subset of the five security-sensitive records as inputs (with no padding added).

All of the tested rootkits were introduced into the kernel as kernel modules. As a result, randomizing a single record type named “module” defeated all tested rootkits before they were loaded into the kernel. The specific failure was that the module loading procedure could not find the module’s name in the attacker-provided module struct.

It is possible for malicious kernel code to be introduced through means other loadable kernel modules [1]. We performed some further experiments without randomizing the “module” record type to test our protection mechanism against rootkit behaviors. We found that if we randomize the record type named “task_struct,” we prevent hp, adore-ng, hideproc from hiding system processes. If we randomize the record type named “module,” we prevent hidefile from hiding system files.

Similar to our evaluation of RFOR, to evaluate the security benefits of subroutine argument order randomization, we employed our SAOR plug-in to compile Linux kernel version 3.2.46 with all calls-to, declarations-of, and types named “pid_task” randomized. This kernel subroutine returns the corresponding task_struct for a given process identifier (PID). We then attempted to use the hp rootkit against this kernel. The hp rootkit was not able to hide a process as a result of SAOR randomization. Though the rootkit loaded successfully into kernel space, its payload was neutralized.

4.5.2 Randomizability Analysis

We used our RFOR Fitness plugin to test the suitability of the record type “task_struct” for randomization. For each potentially unsafe condition found in the source code, our plugin emits a warning message complete with file name and line number[7].

[7] In some cases an approximate location is provided.

For our experiments we compiled the Linux kernel version linux-2.6.38.8 using our RFOR Fitness plug-in with "task_struct" as an argument and the linuxconf configuration template named "allnoconfig."

The plug-in reported that there were 312 conditions found in the kernel source code that may disqualify task_struct from randomization. After careful analysis we found that many of the cases were not actually a problem for randomization and could be automatically detected. We discovered that we could employ whitelists to reduce the total number of false-positive. Following is a list of heuristics that we used to create the whitelists. These heuristics are provided as an aid to analysis. The specifics, e.g. function names and types, will vary from system to system. The RFOR Fitness plug-in results for task_struct both with and without whitelists are provided in Table 4.1.

Generics

Often large code bases provide a reusable set of generic types, macros, and functions that implement common data structures such as lists and queues. These data structures can be reused for a variety of types. For example, the Linux process list is a list of task_structs. The same list implementation used for processes, can be used to create other kinds of lists. To allow the list to hold multiple types, at some point the type will be untyped and cast as a generic list member. This casting trips the assignment to/from $\tau *$ condition.

We found that we could whitelist specific Linux types and drastically reduce the number of conditions reported by the RFOR Fitness plug-in. Specifically we whitelisted the following pointer types for assignment: list_head, hlist_node, rcu_head, plist_node, sched_entity, sched_rt_entity, __wait_queue.private, raw_spinlock.owner, mid_q_entry.callback_data. Additionally, we whitelisted the function named heap_insert[8].

[8] Many data structure operations were macros. This function was an exception.

In addition to generic data structures, there were three generic error-related functions we determined to be safe for field reordering. Specifically we whitelisted the following functions: IS_ERR, PTR_ERR and ERR_PTR.

Memory Allocation Functions

When memory is dynamically allocated the raw memory begins untyped and is cast as a type. The assignment of this untyped memory to a $\tau *$ is reported by our RFOR Fitness plug-in. We found that we could whitelist specific Linux memory allocation function and reduce the number of conditions reported by the RFOR Fitness plug-in. Specifically we whitelisted all of the malloc functions detected automatically by GCC using the DECL_IS_MALLOC macro in addition to the kmem_cache_alloc and kmem_cache_free functions.

Address of Field Conversion

Our plugin reports when the address of a field of τ is taken. This is a frequent occurrence. As mentioned in Section 4.3.2, this is likely not problematic for field reordering. However, our plug-in detects the condition. To aid in analysis, we whitelist the condition where the address of the field is taken but not immediately converted into an integer.

The problematic case for field reordering is when the address of a field is used to calculate the relative position of a sibling field. To do this math, the address would have to be converted into an integer eventually. Again we emphasize that this kind of calculation has not been observed in source code, is difficult to get correct because of compile-time field realignment, and would make the source code difficult to maintain; yet, it remains a possibility. Our plug-in can be configured to report all instances if desired.

Table 4.1
RFOR Fitness Report for task_struct

Test	**W/O Whitelists**	**W/ Whitelists**
Assignment To task_struct *	52	1
Assignment From task_struct *	46	2
Address Taken of task_struct Field	214	0
	312	3

Table 4.2
Performance impact of RFOR

Phoronix Benchmark	**w/o RFOR**	**w/ RFOR**
Gzip Compression	36.12s	35.97s
Timed ImageMagick Compilation	231.14s	232.46s

Table 4.3
Time to compile kernel

Options	**Avg Time**
w/o Plug-Ins	86.730s
w/ RFOR Plugin	87.9087s
w/ Fitness Plugin	87.2170s
w/ SAOR Plugin	87.8303s

4.5.3 Performance

We used the Phoronix test suite to measure the system performance both with and without the record randomization.The results are shown in Table 4.2. With a single record randomized, task_struct, randomization has no observable performance impact.

To test the performance impact of our compiler plug-in's on compilation time, we used the "time" command to measure compilation time with each plug-in. For each test we compiled a minimal Linux kernel version 2.6.38.8 using the configuration template named "allnoconfig." We ran each compilation test three times and calculated the average. The percentage increase of compilation time was between 0.5% and 1.35%. The average results are shown in Table 4.3[9].

[9] For our tests, task_struct was not nested in a union therefore the related test results are excluded

4.6 Discussion

The record field-order randomization suitability analysis method described in Section 4.3.2 applies directly to the C programming language. C is the primary language used for commodity operating system kernels (NT, XNU and Linux). Kernels written in a different programming language may use additional language features that are incompatible with field order randomization.

The conservative static RFOR suitability analysis technique described in Section 4.3.2 is likely to report false-positives, that a record type is not randomizable when in actuality it is safe for randomization. We introduced a heuristics based approach in Section 4.5.2 to aid in analysis of the the Linux kernel specifically. This approach could be refined and made more general by adding pointer analysis.

We assume that the rules described in Section 4.3.2 are comprehensive for RFOR suitability analysis. Our analysis of source code, survey of the literature including [70], and experimentation leads us to the conclude that the ruleset is complete. If our ruleset is incomplete, RFOR suitability analysis would omit conditions that should be reported (false-negatives).

According to the C11 standard, a single structure may have at most 1023 members and a single function definition may have at most 127 parameters [75]. To conform to the standard and the capabilities of standards-compliant compilers, RFOR and SAOR may not add counterfeit padding that exceeds these limits. This reduces, particularly in the case of SAOR, the total number of permutations possible for each randomization technique. These limitations therefore reduce the efficacy of our techniques against brute-force attack.

Many architectures including ARM, X86-64, MIPS and SPARC have argument registers; function arguments may be store in processor registers rather than on the stack. Figure 4.2 illustrates a stack-based argument passing paradigm. However, the design of SAOR is not dependent on stack-based argument passing because reordering occurs in the AST and yielded machine code will be reordered regardless of the argument storage paradigm.

If an attacker were to gain access to the randomized executable file or to the randomized binary image in memory, he may be able to reverse engineer the randomized layout order and use the information to customize an attack. Randomized program files should be protected. Additionally, special memory devices, such as /dev/kmem on Linux, should be disabled and other kernel protection mechanism, such as those described in Chapter 3 should be employed to protect the in-memory randomized code.

Our randomization techniques likely have other benefits not addressed herein. For example, similar to our minimization technique, described in Chapter 5, randomization may assist in protecting systems against ROP attacks. ROP attacks depend considerably on the memory layout of the victim programs. If the program is randomized using our techniques, then ROP attacks may be subverted.

Both randomization techniques may complicate program debugging. If debug information is included in the yielded executable file, the debugging information will correspond to either the randomized layout or the unrandomized source-order layout. In the former situation, the debugging information could potentially be used by an attacker to acquire information about the layout. In the latter situation, the debugging information would be incorrect. For our implementation, if debug information is stored, it is stored prior to layout randomization.

4.7 Summary

In conclusion, we have demonstrated that memory layout randomization is an effective defense against kernel rootkits and that these compile-time techniques incur no run-time overhead making them suitable for even low-powered devices. Further, we have demonstrated that our static analysis technique is an effective way to automatically determine the suitability of a record for field order randomization.

5 ENSURING THE MINIMALITY OF INCLUDED KERNEL COMPONENTS

5.1 Introduction

Code injection prevention and authentication techniques, such as those described in Chapter 3, are still vulnerable to return and jump oriented programming (ROP and JOP respectively) attacks because the payload executable code is already present in kernel memory. The instructions are merely reused in unintended and malicious ways. With large commodity operating system, the amount of reusable code is abundant.

As described by Bryant et. al. in their work on Poly^2, the operating system kernel often includes code that is unnecessary for the applications that are running on the system [54]. General-purpose operating system vendors include kernel code for many possible hardware profiles and system use cases to cover the most uses with a single piece of software. Though this makes the kernel executable larger, it reduces the number of versions that the vendor must support and update. This extra code, though perhaps convenient for both end-users and vendors is a security liability.

The seminal work on return-oriented programming is titled, in-part, "innocent flesh on the bone," a crude reference to the code flesh left on the bone, the kernel in our case, and available for devouring by attackers [3]. In Poly^2, researchers remove unnecessary code at compile time. Though there has been work done in adding code to a kernel at run-time, such as loadable kernel modules, there has not been work done in removing code from a kernel at run-time. In our work we intend to garner the benefits of the kernel reduction technique described in Poly^2 at run-time and reduce the vulnerability of commodity operating system kernels.

Our hypothesis is that it is possible to improve the security of a kernel against return and jump oriented programming attacks by deactivating extraneous kernel code at run-time, thereby limiting the supply of reusable instructions that can be used to construct return-oriented gadgets.

To test our hypothesis, we introduce two novel techniques for run-time kernel minimization. The first is an out-of-the-box function eviction technique. The second is a kernel-based non-executable page technique. We implement a prototype for the out-of-the-box technique and report the results.

5.2 Related Work

Work has been done in design-time techniques for minimizing the code contained in a kernel. Minix and Mach are microkernel designs that attempt to include only necessary components in the kernel space [50, 76].

Work has been done in compile-time techniques for reducing the code contained in a kernel. In the Poly^2 framework, kernels are minimized at compile-time based on the kernel mechanisms needed for specific applications to improve security [54]. For the operating system family named Choices, the kernel is minimized at compile-time based on the functions needed for a specific embedded system to make the kernel as small as possible [55].

Work has been done in run-time techniques for optimizing the code contained in a kernel. The Synthesis kernel optimizes system functions at run-time that get reused frequently to improve system performance [77].

5.3 Chapter Organization

The remainder of this chapter is organized as follows: we begin in Section 5.4 by laying out the design of our minimization techniques followed by a description of our KIS implementation in Section 5.5. In Section 5.6, we evaluate the security merits of our approaches. Finally, in Section 4.7, we summarize our contributions and findings.

5.4 Design

5.4.1 Problem

Modern commodity operating systems are equipped with monolithic kernels that, by design, contain code that does not strictly need kernel-level privileges. Since 1991 the Linux kernel has grown in size from 10,000 lines of code (LOC) to 15,004,006 LOC in 2012. From 2010 to 2012 it more than doubled in size. Likewise, the Windows NT Kernel has 50 million LOC by some estimates [1]. This excessive code increases the risk of kernel-level exploitation.

We are motivated to reduce this risk by dynamically applying the *economy of mechanism* to the operating system kernel code; to prune the kernel back to the minimum amount of instructions that are required for each specific system use case and hardware configuration at run-time. To keep with the carnivorous analogy, we intend to remove as much kernel flesh from the bone as possible. We refer to this technique as *run-time kernel minimization*.

5.4.2 Approach

In this section we describe the general approach to run-time kernel minimization. We introduce a run-time kernel specialization *security monitor* named KIS. KIS is an acronym borrowed, in-part, from a common design principle named KISS that stands for "keep it simple, stupid." The economy of mechanism security principle can be described colloquially as the KISS principle. For our use, KIS stands for "kernel instructions specialization" *and* "keep it simple." KIS is the mechanism that deactivates and activates code at run-time.

[1] http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/

Code Deactivation

We have devised two general techniques for deactivating instructions at run-time. The first is function-level code deactivation. The second is page-level code deactivation.

The function-level deactivation technique modifies kernel functions that are resident in memory. Offline, the kernel is analyzed and a profile is generated. The profile is a list of pairs. The first element in each pair is the address of a kernel function. The second element in each pair is the size of the function in bytes. Each pair can be used to define the byte range of every kernel function.

Online, after the kernel has been loaded into memory, for each function that must be deactivated, we will refer to these functions as mutant functions. KIS replaces the mutant function body with alternative instructions. The form of the alternative instructions depends on whether or not the mutant function can be restored to its original. If the mutant function must never be restored, then the mutant function may be filled with a return instruction, a NOP sled, and a final return instruction. If instructions are fetched from anywhere in the mutant function, execution will immediately return to the caller. Depending on the callee-caller contract, some register restoring may also be necessary. As described in [18], the return value of "-1" can be used as an impostor return value that is handled by some caller code. If the mutant function should truly never be used by the system, then the mutant function will never be called legitimately and illegitimate calls will fail gracefully in many cases.

If mutant function restoration is allowed, then the mutant body is replaced with an interrupt instruction (INT3) or series of interrupt instructions that, if executed, would pass control to an interrupt handler. In the case that KIS is located "out of the box" in a hypervisor, the interrupt would be raised to the hypervisor first and KIS could manage the restoration of the mutant. In the case that KIS is located in the kernel, KIS would have to be installed as an interrupt handler.

In both cases, KIS removes the original mutant function bodies. Any return-oriented programming attacks that rely on the original mutant body will hampered, if not completely prevented.

The second code deactivation technique is similar to the NICKLE-KVM approach described in Chapter 3. Rather than deactivating the code at the function-level, we deactivate code at the page-level. If a page contains code that must be deactivated, we call this a mutant page, then the corresponding page table entry is set to non-executable. If instructions located in that page are fetched, control would then be passed to KIS by way of a page fault exception.

Because the page-level deactivation is less granular than the function-level technique previously described, special consideration must be given to pages that contain both code that should be executable and code that should not be executable. If such mixed pages exist, they should likely remain executable. If KIS is located in the hypervisor, then the mixed mutant pages may be set non-executable. This would reduce the size of the kernel until the page-adjacent approved code was activated thereby activating the excessive code at the same time.

The Poly^2 approach is to remove unnecessary code once and never reintroduce it. This occurs at compile time. If an analogous run-time approach is used, then we would need to calculate offline the kernel code required by the applications that will be running on the system and deactivate the unneeded instructions at run-time. An alternative approach is to deactivate code liberally, leaving only essential components activated, and reactivate code on an as-needed basis. Deciding what code to reactivate is described later.

KIS Security Monitor

The purpose of the security monitor is to deactivate and reactivate code and log related events. KIS may be activated by system exceptions such as a page fault or debug interrupt previously described or by a message passing. Additionally, it may itself be monitoring the system for specific conditions and trigger code to be activated or deactivated based on predefined events.

The security monitor can be used passively for anomaly *detection*. If the deactivated code should not be executed but is, then the event is logged and used as part of an intrusion detection analysis. It is passive in that, in this mode, it has a permissive re-inclusion policy that allows for the code to be reactivated. In this mode KIS would act primarily as a detection mechanism rather than a prevention mechanism.

KIS may also take a more active role. If the deactivated code should not be executed but is, then the event is logged and used as part of an intrusion detection analysis. If no reactivation policy is available, then KIS would terminate the process. If a reactivation policy is available that permits reactivation, then the security monitor reactivates the code.

Various deactivation/reactivation policies may be constructed. We have devised two examples of such policies. The first is based on an event model; the second on a control flow graph technique such as the one described by [35].

One example of an event driven policy is the following: for a kernel that supports LKM, at system boot many modules are loaded automatically to service the hardware present in the system. Suppose that this system is a server system in a data center that never needs any new hardware hot-plugged. Once the system is booted, the module loading code can be deactivated. A “modules loaded” message could be passed to KIS or KIS could be configured to detect that event and deactivate the module loading code.

A second example of a reactivation policy is *on-demand* activation. Suppose that after system boot-up is complete all non-essential kernel code is deactivated. The code is then reactivated on an as-needed basis. A permissive approach is to reactivate code on-demand. In this scenario, KIS verifies that the first instruction fetched from the mutant, function or page, is the first instruction byte of a function body.

Though permissive, this is a minimally intrusive approach that allows the system to run nearly unfettered while still preventing ROP and JOP attacks from reusing instructions originally found in the mutants. In the case that the kernel memory is not visible to the attacker, for example /dev/kmem access is prevented in the Linux kernel, then the ROP and JOP attacks would be formulated against the offline version of the kernel code, the code that contains all of the original mutant function instructions. Online, any attack that depends on a mutant instruction would fail. This approach is described further and evaluated in Sections 5.5 and 5.6.

The previously described permissive on-demand activation policy can be constrained further using control-flow analysis. Offline, a control-flow graph is generated for the kernel. The graph describes all parent-child relationships between code regions and all valid control flow entry points. For example, if code is deactivated at the function level, then the first time that code is fetched, it should be on a function boundary and at least one parent function should already be activated. If the code is deactivate at the page level, then the entry control-flow validation would work similarly. However, for code that crosses page boundaries, additional entry points would be required.

KIS Location

The kernel is intended to run with all of its instructions intact. If essential instructions are removed, then a kernel fault will likely destabilize if not destroy the kernel operation. If, for example, KIS is placed in the system kernel, the page fault interrupt handler must be able to service page faults at all times. If KIS is placed outside of the system and code may be reactivated on demand, then the number of essential components may be minimal.

Out-of-the-box security mechanisms, such as those described in Chapter 3, are tamper-resistant against guest-based attacks. In-kernel security mechanisms are subject to vulnerabilities of the kernel that it is trying to protect. In situations where virtualization is is used, such as in a cloud hosting environment, KIS should be located in the hypervisor for maximum security benefits. However, in situations where virtualization is not appropriate, such as resource constrained mobile devices, KIS should be located in the kernel.

Out-of-the-box security mechanisms must be designed carefully as not to impose too many virtual machine exits (VM exits). VM exits are computationally expensive. One advantage to an in-kernel KIS design is that it can be more intrusive and code can be deactivated on a per process basis.

For example, consider the following process-level deactivation scenario. Suppose a sandboxed shell process (SSHELL) is used for launching all processes subjected to kernel minimization. The SSHELL has all non-essential kernel code pages marked as non-executable. When a process is created via a fork, the virtual memory page table is copied but the physical memory is not, thereby propagating the non-executable kernel code pages to all child processes. The KIS-enabled kernel has a modified page-fault handler. When an instruction is fetched from a mutant page, the KIS-assisted page-fault handler will detect if the fault was caused by an NX permissions violation. If so, then KIS will determine whether or not to allow the page to be reactivated based on some previously described reactivation policy. If allowed, then the mutant page is reactivated and the page table entry is set to executable for the duration of the process and all spawned processes. All processes, such as threads and spawned processes, that share this process's page table have the same activated kernel code. Unlike previously described designs, including Poly^2 that are system-level, this design is process-level; though the kernel is mapped into each process, SSHELL ancestor processes will not have access to all of the kernel code.

5.5 Implementation

We implemented our run-time kernel minimization security monitor, KIS, using an Intel i5-2410M 2.30GHz processor. Both our host and guest systems run an *unmodified* version of Ubuntu (11.04 2.6.38-8-generic). The host runs the 64 bit version of Ubuntu (x86_64) and the guest runs the 32 bit version (i686). We added our security monitor to the Linux Kernel Virtual Machine (KVM) version kvm-kmod-2.6.38-rc7.

Our run-time kernel minimization implementation closely resembles the permissive on-demand method described previously in Section 5.4.2. KIS is placed in a Linux KVM hypervisor and protects a single guest running as a QEMU/KVM virtual machine. Guest instructions are deactivated at the function level. Functions are reactivated when its first function instruction is executed. Though this technique is permissive, it prevents ROP and JOP that reuse instructions directly preceding free-branch instructions such as return instructions. Such instructions, that are suitable for reuse, do not occur in the function preamble. As a result ROP and JOP gadgets will *not* naturally trigger on-demand function activation.

KIS uses virtual machine introspection to intercept specific guest events. The first event intercepted is the completion of system startup. For the Linux kernel, we purposefully chose a point in the boot sequence when the kernel is fully loaded into memory and has already patched itself. This occurs right before the kernel function named init_post is executed. We refer to this event as the INIT event. The functions that are executed prior to the INIT event are not protected. Many of those functions are part of the ELF file section named .init.text and are deallocated shortly after booting completes and are therefore likely not part of an ROP/JOP attack.

When the INIT event occurs, our security monitor takes control. It reaches into the virtual machine and deactivates *all* kernel functions. To deactivate functions it replaces the first byte of every function with a INT3 byte (0xCC). When the INT3 byte is executed by the guest at some later time, it causes a software interrupt to occur. This interrupt is raised to the hypervisor and our security monitor handles the interrupt. If the interrupt is caused by the first byte of a function, then the original byte is replaced. Because INT3 is an interrupt and not an exception, when the virtual machine resumes, it tries the instruction again. This time however, the original instruction is in place and the function executes normally. For this prototype the function remains activated for the duration of the system.

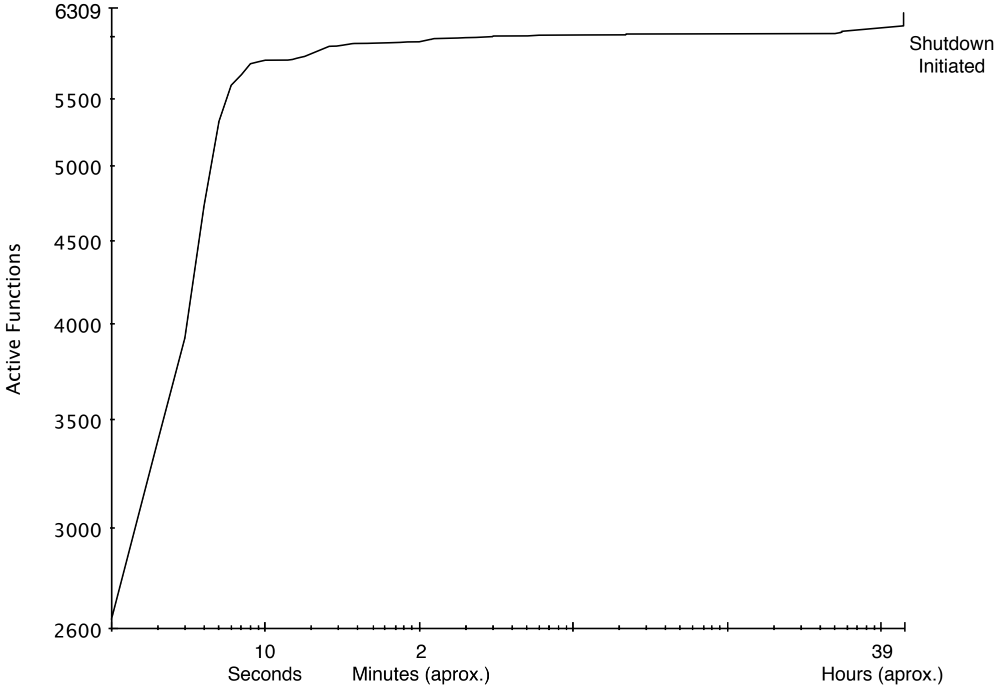


Figure 5.1. Reactivation of functions over time (log scale)

To evaluate our design, we configured the protected guest with a common LAMP web server stack (Linux, Apache, MySQL, and PHP). We installed all of these packages from the Ubuntu packaging system. To emulate common usage, we configured and installed a popular blogging web application named WordPress and configured all relevant LAMP modules.

After configuring the web server, we rebooted the system with KIS protection enabled. At boot, KIS disabled 28,828 kernel functions, and 94% of the functions that were eventually reactivated were activated within the first 1 minute after deactivation. We allowed the web server to run for 39 hours. Upon shutdown, 160 functions were activated. A plot of these numbers is illustrated in Figure 5.1.

Of the 28,828 functions deactivate, only 6,309 were reactivated. 78% of the kernel functions were not needed by this web server workload. 72% of the instruction bytes were not needed by this web server workload. KIS can therefore effectively remove nearly 3/4 of the instructions that could be reused by ROP and JOP attacks.

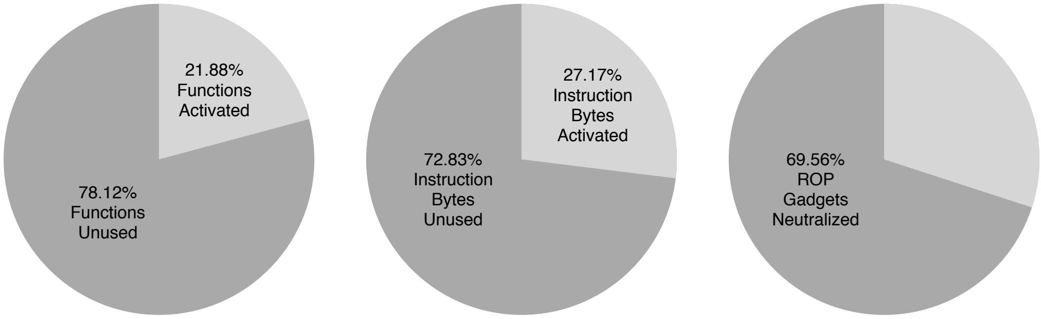


Figure 5.2. KIS keeps it simple

To evaluate the effectiveness of KIS against attack, we used an ROP exploitation tool named ROPgadget v4.0.3 [78]. ROPgadget found 92 reusable gadgets in the Linux kernel ELF file (vmlinux-2.6.38-8-generic). For our evaluation, with 6,309 kernel functions activated, KIS would prevent 69.56% of the gadgets from executing. Any attack that depends on one of the 64 deactivated gadgets will fail to run on our web server's kernel. Figure 5.2 illustrates these numbers.

5.7 Discussion

The on-demand function activation technique described in Sections 5.4.2 and 5.5 assumes that all valid calls to a function, set the program counter to the address of the first instruction byte for the function. As a result features such as function nesting, a non-standard C feature found in GCC, are not supported unless the parent function is first activated [75]. Similarly, according to the C standard, nonlocal jumps from a callee function are permitted when the jump destination is located in the calling function. If C programming standards are followed, the target function of a nonlocal jump will have already been previously activated. It is possible however, that valid machine code could be created that defies our assumptions and that valid programs would be prevented from executing as intended as a result of KIS.

We have demonstrated that it is possible to reduce the kernel code significantly for one specific well-defined workload. We have demonstrated that for this specific workload and for one known set of ROP exploits, that kernel security is improved. It is possible that some workloads will activate significantly more of the kernel code. It is also possible that other ROP techniques may produce significantly more attack possibilities.

If an attacker understood and detected the on-demand function activation technique described in Chapter 5, he may be able to directly or indirectly cause valid calls to a set of functions that contain code segments necessary for a specific return-oriented attack.

5.8 Summary

We introduced two novel techniques for run-time kernel minimization. We showed that these techniques can be an effective defense against kernel-based ROP attacks. Further, we demonstrated that for a common Linux web server workload, the distributed, unmodified Linux kernel tested was far too large; the workload demanded only 27.17% (by bytes) of the shipped kernel code to run.

Code injection prevention and authentication techniques provide a strong defense against attacks that require foreign code to run in kernel space. However, these techniques are insufficient on their own to defend against attacks that reuse existing kernel code. Combining the former with run-time kernel minimization significantly improves the security of a commodity operating system kernel.

6 CONCLUSIONS

6.1 Summary

In this dissertation we have shown that it is possible to strengthen the defenses of commodity, general-purpose, computer operating systems by increasing the diversity of, validating the integrity of, and ensuring the minimality of the included kernel components without modifying the kernel source code. Such protections can therefore be added to existing widely-used unmodified operating systems to prevent malicious software from executing in supervisor mode.

6.1.1 Validation

In Chapter 3 we introduced a novel technique to *validate the integrity* of unmodified, self-patching, commodity, general-purpose computer operating system kernels. We implemented a code injection prevention technique that relies upon kernel code authentication. We discovered that some modern kernels are "self-patching;" the kernel instructions mutate at run-time. Previous cryptographic hash validation procedures were not able to handle such modifications. In addition to implementing an out-of-the-box code injection prevention mechanism that takes advantage of hardware virtualization for significantly increased performance, we designed and implemented a system that validates the integrity of each instruction introduced by a self-patching kernel at run-time and demonstrated that patch-level validation correctly permits valid kernel patches to be applied and rejects patches that are invalid.

6.1.2 Diversification

In Chapter 4, we introduced a novel technique to *increase the diversity* of commodity, general-purpose, computer operating system kernels. We introduce a new method for randomizing the stack layout of function arguments. We refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record. We showed that our static analysis technique is an effective way to automatically determine the suitability of a record for field order randomization. Additionally, we showed that by strategically selecting just a few components for randomization, our techniques prevent all tested kernel rootkits. These compile-time techniques incur no run-time overhead and makes them suitable for even low-powered devices.

6.1.3 Minimization

In Chapter 5 we introduced a novel technique to *ensure the minimality* of unmodified, commodity, general-purpose, computer operating systems kernels. We show that it is possible to improve the security of a standard commodity operating system kernel against return oriented programming attacks by deactivating extraneous kernel code at run-time and thereby limiting the supply of reusable instructions that can be used to construct return-oriented gadgets. We designed and implemented our run-time kernel minimization and demonstrated that, for a common web server workload, 72% of the instructions included in the kernel were extraneous and that run-time kernel minimization reduced the number of ROP usable gadgets found in the kernel by 70%.

6.2 Future Work

Mobile computing is rapidly replacing desktop and server computing. Many of the techniques described in this dissertation rely upon virtualization. The compile-time randomization techniques described in Chapter 4 impose no run-time overhead that is essential for low-powered devices. To bring these techniques to mobile computing, we must investigate how to apply the principles discovered to a mobile context. Can we run a minimal hypervisor on mobile devices? For the mobile devices that rely heavily upon a Java Virtual Machine (JVM), such as Android, can we reuse the JVM for code injection prevention, code authentication, and code minimization? In the case of run-time kernel minimization, can the in-kernel design be used?

We do not believe that general-purpose operating system kernels can continue to grow at the current rate. It is too risky. The run-time kernel minimization technique described here is a patching mechanism for large multipurpose kernels. It would be better to redesign kernels to include a finer-grained code deactivation paradigm that allows vendors the flexibility to ship one software artifact for many use cases but also allows the system to *keep it simple*.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Elias Levy. Smashing the stack for fun and profit. *Phrack Magazine*, 49:14–16, August 1996.

[2] Sebastian Krahmer. X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/ krahmer/no-nx.pdf, 2005.

[3] 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*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[4] Michel Kaempf. Vudo malloc tricks. *Phrack*, 57, 2001.

[5] scut. Exploiting format string vulnerabilities. Technical report, Team TESO, 2001.

[6] blexim. Basic integer overflows. *Phrack*, 60, 2002.

[7] Oded Horovitz. Big loop integer protection. *Phrack*, 60, December 2002.

[8] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Andi Kleen. Runtime memory barrier patching. Linux-Kernel Mailing List, April 2003.

[10] J.-M. De Goyeneche and E.A.F. De Sousa. Loadable kernel modules. *Software, IEEE*, 16(1):65–71, 1999.

[11] J. von Neumann. First draft of a report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4):27 –75, 1993.

[12] *Intel 64 and IA-32 Architectures. Software Developer's Manual*, volume 3a: system programming guide, part 1 edition, May 2011.

[13] *AMD64 Architecture Programmer's Manual*, volume 2: system programming edition, September 2012.

[14] David L. Weaver and Tom Germond. *The SPARC Architecture Manual.* SPARC International, Inc., San Jose, California, version 9 edition, 1994.

[15] John Holland. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. In *Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 108–113, New York, NY, USA, 1959. ACM.

[16] E. W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1988.

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

[19] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. Top 25 most dangerous software errors. Technical report, CWE/SANS, 2011.

[20] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Computer Communication Review*, 19(1):17–57, January 1989.

[21] Solar Designer. Getting around non-executable stack (and fix). Bugtraq mailing list, August 1997.

[22] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 58, December 2001.

[23] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[24] Edsger W. Dijkstra. The structure of the the-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[25] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, March 1972.

[26] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308, sept. 1975.

[27] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? Technical report, Western Research Laboratory, 1989.

[28] *Intel 64 and IA-32 Architectures. Software Developer's Manual*, volume 3b: system programming guide, part 2 edition, May 2011.

[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[30] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Operating Systems Review*, 41:335–350, October 2007.

[31] James P. Anderson. Computer security technology planning study, volume 1. Technical report, Air Force Systems Command, 1972.

[32] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

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

[34] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. http://www.cs.ucdavis.edu/research/tech-reports/1996/CSE-96-11.pdf, 1996.

[35] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):4:1–4:40, November 2009.

[36] Dannie Michael Stanley, Zhui Deng, Dongyan Xu, Rick Porter, and Shane Snyder. Guest-transparent instruction authentication for self-patching kernels. *Proceedings of Military Communications Conference (MILCOM)*, October 2012.

[37] Solar Designer. Non-executable user stack. http://www.openwall.com/linux/, August 1997.

[38] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[39] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[40] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[41] PaX Team. Address space layout randomization. Technical report, Open Source Security, Inc., 2003.

[42] 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*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[43] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 107–126, Berlin, Heidelberg, 2009. Springer-Verlag.

[44] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In Atul Prakash and Indranil Sen Gupta, editors, *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin / Heidelberg, 2009.

[45] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[46] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[47] R. R. Schell. Dynamic reconfiguration in a modular computer system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1971.

[48] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 145–156, New York, NY, USA, 1996. ACM.

[49] C. DiBona and S. Ockman. *Open sources: Voices from the open source revolution.* O'Reilly Media, Incorporated, 1999.

[50] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[51] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[52] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[53] Neal H. Walfield and Marcus Brinkmann. A critique of the GNU hurd multi-server operating system. *SIGOPS Oper. Syst. Rev.*, 41(4):30–39, July 2007.

[54] E. Bryant, J. Early, R. Gopalakrishna, G. Roth, E.H. Spafford, K. Watson, P. William, and S. Yost. Poly2 paradigm: A secure network service architecture. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 342 – 351, dec. 2003.

[55] Roy Campbell, Garry Johnston, and Vincent Russo. Choices (class hierarchical open interface for custom embedded systems). *SIGOPS Operating Systems Review*, 21(3):9–17, July 1987.

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

[57] Jonathan Corbet. Jump label. Technical report, Linux Weekly News, October 2010.

[58] Steven Rostedt. Mcount tracing utility. Linux-Kernel Mailing List, January 2008.

[59] Rusty Russell. X86 paravirt_ops: Binary patching infrastructure. Linux-Kernel Mailing List, August 2006.

[60] Sudhanshu Goswami. An introduction to KProbes. Technical report, Linux Weekly News, April 2005.

[61] Dannie Michael Stanley, Dongyan Xu, and Eugene Spafford. Improved kernel security through memory layout randomization. *Proceedings of International Performance Computing and Communications Conference (IPCCC)*, December 2013.

[62] Paul Klemperer. Markets with consumer switching costs. *The Quarterly Journal of Economics*, 102(2):375–394, 1987.

[63] R. Anderson. Why information security is hard-An economic perspective. In *Proceedings of the 17th Annual Computer Security Applications Conference*, ACSAC '01, pages 358–, Washington, DC, USA, 2001. IEEE Computer Society.

[64] Pei-Yu Chen, Gaurav Kataria, and Ramayya Krishnan. Software diversity for information security. In *Workshop on the Economics of Information Security (WEIS), Harvard University, Cambridge, MA*, 2005.

[65] Daniel Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P Pfleeger, John S Quarterman, and Bruce Schneier. Cyberinsecurity: The cost of monopoly. *Computer and Communications Industry Association (CCIA)*, 2003.

[66] Eugene H. Spafford. Computer viruses as artificial life. *Artificial Life*, 1(3):249–265, January 1994.

[67] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January 1997.

[68] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[69] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, 34(5):13–24, May 1999.

[70] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 233–244, Washington, DC, USA, 2006. IEEE Computer Society.

[71] Free Software Foundation, Inc. *GNU Compiler Collection (GCC) Internals*, 4.9.0 edition, 2013.

[72] Semyon Boyko. Driver to hide files in Linux OS. http://www.codeproject.com/Articles/444995/Driver-to-hide-files-in-Linux-OS, August 2012.

[73] gadgetweb.de. How to: Hijacking the syscall table on latest 2.6.x kernel systems. http://gadgetweb.de/linux/40-how-to-hijacking-the-syscall-table-on-latest-26x-kernel-systems.html, June 2010.

[74] Junghwan Rhee, Zhiqiang Lin, and Dongyan Xu. Characterizing kernel malware behavior with kernel data access patterns. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 207–216, New York, NY, USA, 2011. ACM.

[75] International Organization for Standardization and International Electrotechnical Commission. 9899:201x programming languages – C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, April 2011.

[76] David Golub, Randall Dean, Alessandro Forin, All Dean, Ro Forin, and Richard Rashid. Unix as an application program. In *In USENIX 1990 Summer Conference*, pages 87–95, 1990.

[77] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[78] Jonathan Salwan and Allan Wirth. ROPgadget. Technical report, Shell-Storm, March 2011.

VITA

VITA

Dannie M. Stanley received an M.S. in computer science from Ball State University in July of 2008. He entered Purdue University in the fall of 2008. He obtained a Ph.D. in December of 2013 under the direction of Professor Eugene Spafford and Professor Dongyan Xu. During his time at Purdue, Dannie served as the president of two student organizations: Upsilon Pi Epsilon: International Honor Society for the Computing and Information Disciplines and the Purdue CERIAS Student Association. In the spring of 2013, he was awarded the CERIAS Diamond Award For Outstanding Academic Achievement. In the fall of 2013, he joined the faculty of Miami University in Oxford Ohio as a visiting instructor in the department of Computer Science and Software Engineering.