Return-Oriented Rootkits:
Bypassing Kernel Code Integrity Protection Mechanisms

Ralf Hund Thorsten Holz Felix C. Freiling

Laboratory for Dependable Distributed Systems
University of Mannheim, Germany
hund@uni-mannheim.de, {holz,freiling}@informatik.uni-mannheim.de

Abstract

Protecting the kernel of an operating system against attacks, especially injection of malicious code, is an important factor for implementing secure operating systems. Several kernel integrity protection mechanism were proposed recently that all have a particular shortcoming: They cannot protect against attacks in which the attacker re-uses existing code within the kernel to perform malicious computations. In this paper, we present the design and implementation of a system that fully automates the process of constructing instruction sequences that can be used by an attacker for malicious computations. We evaluate the system on different commodity operating systems and show the portability and universality of our approach. Finally, we describe the implementation of a practical attack that can bypass existing kernel integrity protection mechanisms.

1 Introduction

Motivation. Since it is hard to prevent users from running arbitrary programs within their own account, all modern operating systems implement protection concepts that protect the realm of one user from another. Furthermore, it is necessary to protect the kernel itself from attacks. The basis for such mechanisms is usually called *reference monitor* [2]. A reference monitor controls all accesses to system resources and only grants them if they are allowed. While reference monitors are an integral part of any of today's mainstream operating systems, they are of limited use: because of the sheer size of a mainstream kernel, the probability that some system call, kernel driver or kernel module contains a vulnerability rises. Such vulnerabilities can be exploited to subvert the operating system in arbitrary ways, giving rise to so called *rootkits*, malicious software running without the user's notice.

In recent years, several mechanism to protect the integrity of the kernel were introduced [6, 9, 15, 19, 22], as we now explain. The main idea behind all of these approaches is that the memory of the kernel should be protected against unauthorized injection of code, such as rootkits. Note that we focus in this work on kernel integrity protection mechanisms and not on control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms, which are orthogonal to the techniques we describe in the following.

1.1 Kernel Integrity Protection Mechanisms

Kernel Module Signing. Kernel module signing is a simple approach to achieve kernel code integrity. When kernel module signing is enabled, every kernel module should contain an embedded, valid digital signature that can be checked against a trusted root certification authority (CA). If this check fails, loading of the code fails, too. This technique has been implemented most notably for Windows operating systems since XP [15] and is used in every new Windows system.

Kernel module signing allows to establish basic security guidelines that have to be followed by kernel code software developers. But the security of the approach rests on the assumption that the already loaded kernel code, i.e., the kernel and *all* of its modules, does not have a vulnerability which allows for execution of unsigned kernel code. It is thus insufficient to check for kernel code integrity only upon loading.

W⊕X. W⊕X is a general approach which aims at preventing the exploitation of software vulnerabilities at runtime. The idea is to prevent execution of injected code by enforcing the W⊕X property on all, or certain, page tables of the virtual address space: A memory page must never be writable *and* executable at the same time. Since injected code execution always implies previous

instruction writes in memory, the integrity of the code can be guaranteed. The W⊕X technique first appeared in OpenBSD 3.3; similar implementations are available for other operating systems, including the PaX [28] and Exec Shield patches for Linux, and PaX for NetBSD. Data Execution Prevention (DEP) [16] is a technology from Microsoft that relies on W⊕X for preventing exploitation of software vulnerabilities and has been implemented since Windows XP Service Pack 2 and Windows Server 2003.

The effectiveness of W⊕X relies on the assumption that the attacker wishes to modify and execute code in kernel space. In practice, however, an attacker usually first gains userspace access which implies the possibility to alter page-wise permission in the userspace portion of the virtual address space. Due to the fact that the no-executable bit in the page-table is not fine-grained enough, it is not possible to mark a memory page to be executably *only* in user mode. So an attacker may simply prepare her instructions in userspace and let the vulnerable code jump there.

NICKLE. *NICKLE* [19] is a system which allows for lifetime kernel code integrity, and thus rootkit prevention, by exploiting a technology called *memory shadowing*. NICKLE is implemented as virtual machine monitor (VMM) which maintains a separate so-called *shadow memory*. The shadow memory is not accessible from within the VM guest and contains copies of certain portions of the VM guest's main memory. Newly executing code, i.e., code that is executed for the first time, is authenticated using a simple cryptographic hash value comparison and then copied to the shadow memory transparently by NICKLE. Since the VMM is trusted in this model, it is guaranteed that no unauthenticated modifications to the shadow memory can be applied as executing guest code can never access the privileged shadow memory. Therefore, any attempt to execute unauthenticated code can be foiled in the first place. Another positive aspect of this approach is that it can be implemented in a rather generic fashion, meaning that it is perfectly applicable to both open source and commodity operating systems. Of course, NICKLE itself has to make certain assumptions about underlying file format of executable code, e.g., driver files, since it needs to understand the loading of these files. Currently, NICKLE supports Windows 2000, Windows XP, as well as Linux 2.4 and 2.6 based kernels. So far, NICKLE has been implemented for QEMU, VMware, and VirtualBox hypervisors. The QEMU source code is publicly available [20].

The isolation of the VMM from the VM Guest allows for a comparably unrestrictive threat model. In the given system, an attacker may have gained the highest level of privilege within the VM guest and may access the entire memory space of the VM. In other words, an adversary may compromise arbitrary system entities, e.g., files, processes, etc., as long as the compromise happens only *inside* the VM.

SecVisor. SecVisor [22] is a software solution that consist of a general, operating system independent approach to enforce W⊕X based on a hypervisor and memory virtualization. In the threat model for SecVisor an attacker can control everything but the CPU, the memory controller, and kernel memory. Furthermore, an attacker can have the knowledge of kernel exploits, i.e., she can exploit a vulnerability in kernel mode. In this setting, SecVisor "protects the kernel against code injection attacks such as kernel rootkits" [22]. This is achieved by implementing a hypervisor that restricts what code can be executed by a (modified) Linux kernel. The hypervisor virtualizes the physical memory and the MMU to set page-table-based memory protections. Furthermore, SecVisor verifies certain properties on kernel mode entry and exit, e.g., all kernel mode exits set the privilege level of the CPU to that of user mode or the instruction pointer points to approved code at kernel entry. Franklin et al. showed that these properties are prone to attacks and successfully injected code in a SecVisor-protected Linux kernel [8], but afterwards also corrected the errors found.

1.2 Bypassing Integrity Protection Mechanisms

Based on earlier programming techniques like *return-to-libc* [17, 21, 27], Shacham [23] introduced the technique of *return-oriented programming*. This technique allows to execute arbitrary programs in privileged mode without adding code to the kernel. Roughly speaking, it misuses the system stack to "re-use" existing code fragments (called *gadgets*) of the kernel (we explain this technique in more detail in Section 2). Shacham analyzed the GNU Linux libc of Fedora Core 4 on an Intel x86 machine and showed that executing one malicious instruction in system mode is sufficient to construct arbitrary computations from existing code. No malicious code is needed, so most of the integrity protection mechanisms fail to stop this kind of attack.

Buchanan et al. [4] recently extended the approach to the Sparc architecture. They investigated the Solaris 10 C library, extracted code gadgets and wrote a compiler that can produce Sparc machine programs that are made up entirely of the code from the identified gadgets. They concluded that it is not sufficient to prevent introduction of malicious *code*; we must rather prevent introduction of malicious *computations*.

Attacker Model. Like the mentioned literature, we base our work on the following reasonable attacker model. We assume that the attacker has full access to the user's address space in normal mode (local attacker) and that there exists at least one vulnerability within a system call such that it is possible to point the control flow to an address of the attacker's choice at least once while being in privileged mode. In practice, a vulnerable driver or kernel module is sufficient to satisfy these assumptions. Our attack model also covers the typical "remote compromise" attack scenario in network security where attackers first achieve local user access by guessing a weak password and then escalate privileges.

Contributions. In this paper, we take the obvious next step to show the futility of current kernel integrity protection techniques. We make the following research contributions:

- While previous work [4, 21, 23] was based on manual analysis of machine language code to create gadgets, we present a system that fully automates the process of constructing gadgets from kernel code and translating arbitrary programs into return-oriented programs. Our automatic system can use any kernel code (not only libc, but also drivers for example) even on commodity operating systems.

- Using our automatic system, we construct a portable rootkit for Windows systems that is entirely based on return-oriented-programming. It therefore is able to bypass even the most sophisticated integrity checking mechanism known today (for example NICKLE [19] or SecVisor [22]).

- We evaluate the performance of return-oriented programs and show that the runtime overhead of this programming technique is significant: In our tests we measured a slowdown factor of more than 100 times in sorting algorithms. However, for exploiting a system this slowdown might not be important.

Outline. The paper is structured as follows. In Section 2 we provide a brief introduction to the technique of return-oriented-programming. In Section 3 we introduce in detail our framework for automating the gadget construction and translating arbitrary programs into return-oriented programs. We present evaluation results for our framework in Section 4: Using ten different machines, we confirm the portability and universality of our approach. We present the design and implementation of a return-oriented rootkit in Section 5 and finally conclude the paper in Section 6 with a discussion of future work.

2 Background: Return-Oriented Programming

The idea behind a return-to-libc attack [17, 27] is that the attacker can use a buffer overflow to overwrite the return address on the stack with the address of a legitimate instruction which is located in a library, e.g., within the C runtime *libc* on UNIX-style systems. Furthermore, the attacker places the arguments to this function to another portion of the stack, similar to classical buffer overflow attacks. This approach can circumvent some buffer overflow protection techniques, e.g., non-executable stack.

The technique of return-oriented programming was introduced by Shacham et al. [4, 23]. It generalizes return-to-libc attacks by chaining short new instructions streams ("useful instructions") that then return. Several instructions can be combined to a *gadget*, the basic block within return-oriented programs that for example computes the AND of two operands or performs a comparison. Gadgets are self-contained and perform one well-defined step of a computation. The attacker uses these gadgets to cleverly craft stack frames that can then perform arbitrary computations. Fig. 1 illustrates the process of return-oriented programming. First, the attacker identifies useful instructions that are followed by a `ret` instruction (e.g., instruction sequences A, B and C in Fig. 1). These are then chained to gadgets to perform a certain operation. For example, instruction sequences A and B are chained together to gadget 1 in Fig. 1. On the stack, the attacker places the appropriate return addresses to these instruction sequences. In the example of Fig. reffig:rop the return addresses on the stack will cause the executions of gadget 1 and then gadget 2. The stack pointer ESP determines which instruction to fetch and execute, i.e., within return-oriented programming the stack pointer adopts the role of the instruction pointer (IP): Note that the processor does not automatically increment the stack pointer, but the `ret` instruction at the end of each useful instruction does.

The authors showed that both the libc library of Linux running on the x86 architecture (CISC) as well as the libc library of Solaris running on a SPARC (RISC) contain enough useful instructions to construct meaningful gadgets. They manually analyzed the libc of both environments and constructed a library of gadgets that is Turing-complete. We extend their work by presenting the design and implementation of a fully automated return-oriented framework that can be used on commodity operating systems. Furthermore, we describe an actual attack against kernel integrity protection systems by implementing a return-oriented rootkit.

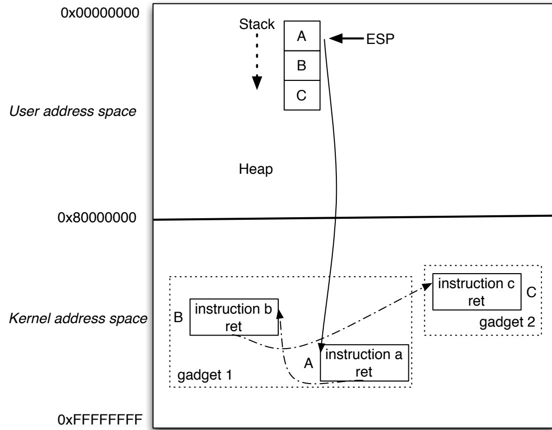

Figure 1: Schematic overview of return-oriented programming on the Windows platform

3 Automating Return-Oriented Programming

In order to be able to create and execute return-oriented programs in a generic way, we created our own, modular toolset which enables one to abstract from the varying concrete conditions one faces in this context. Additionally, our system greatly simplifies the development of return-oriented programs by providing high-level constructs to accomplish certain tasks. Figure 2 provides a schematic overview of our system; it is partitioned into three core components:

- *Constructor*. The Constructor scans a given set of files containing executable code, spots useful instruction sequences and builds return-oriented gadgets in an automatic fashion. These well-defined gadgets serve as a low-level abstraction and interface to the Compiler.
- *Compiler*. The Compiler provides a comparatively high-level language for programming in a return-oriented way. It takes the output of the Constructor along with a source file written in a dedicated language to produce the final memory image of the program.
- *Loader*. As the Compiler's output is position independent, it is the task of the Loader to resolve relative memory addresses to absolute addresses. This component is implemented as library that is supposed to be linked against by an exploit.

All components have been implemented in C++ and currently we support Windows NT-based operating systems running on an IA-32 architecture. In the following paragraphs, we give more details on each component's inner workings.

3.1 Automated Gadget Construction

One of the most essential parts of our system is the *automated construction* of return-oriented gadgets, thus enabling us to abstract from a concrete set of executable code being exploited for our purposes. This is in contrast to previous work [4, 23], which focused on concrete versions of a C library instead. Our system works on an arbitrary set of files containing valid x86 machine code instructions; we will henceforth refer to these files as the *codebase*.

Our framework implements the creation of gadgets in the so-called *Constructor*, which performs three subsequent jobs: First, it scans the codebase to find *useful instruction sequences*, i.e., instructions preceding a return (`ret`) instruction. These instructions can then be used to implement a return-oriented program by concatenating the sequences in a specific way. Our current implementation targets machines running an arbitrary Windows version as operating system and thus we use all driver executables and the kernel as codebase in the scanning phase. In the second step, our algorithm chains the instruction sequences together to form *gadgets* that perform basic operations. We define the term gadget analog to Shacham [23], i.e., gadgets comprise composite useful instruction sequences to accomplish a well-defined task (e.g., perform an AND operation or check a boolean condition). More precisely, when we talk of *concrete gadgets*, we mean the corresponding *stack allocation*, i.e., the contents (return-addresses, constants, etc.) of the memory area the stack register points to. Gadgets represent an intermediate abstraction layer whose elements are the basic units subsequently used by the Compiler for building the final return-oriented program. Gadgets being written to the Constructor's final output file are called *final gadgets*. In the third step, the Constructor searches for exported symbols in the codebase and saves these in the output file for later use by the Compiler.

3.1.1 Finding Useful Instruction Sequences

The first decision that has do be made is describing the basic instruction sequences being the very core of a return-oriented program. As previously mentioned, these instructions occur prior to a `ret` x86 assembler instruction. We have to decide how many instructions preceding a return are considered. For instance, the Constructor might look for sequences such as `mov eax, ecx; add eax, edx; ret`, and incorporate these in the subsequent gadget construction. An easier approach, however, is to consider only a single instruction before a return instruction. Of course, the former attempt has the advantage of being more comprehensive, along with the drawback of requiring additional overhead. This stems from the fact that one has to take every instruction's pos-

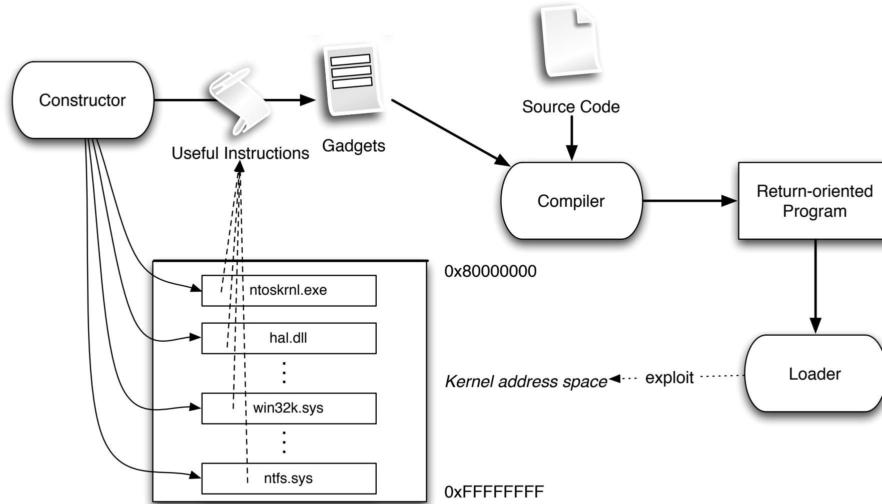

Figure 2: Schematic system overview

sible side-effects on registers and memory into account. In our work, we have thus chosen to implement the latter approach. Rudimentary research has shown that the additional value of using longer instruction sequences hardly justifies the imposed overhead since the effect of the former is not very significant in practice: We have observed that the high density of the x86 instructions encoding does not introduce substantial surplus concerning additional instruction sequences. We would also like to stress that this simplified approach has not turned out to be problematic in our work so far since the codebase of every system we evaluated held sufficient instruction sequences to implement arbitrary return-oriented programs (see Section 4 for details). However, our system might still be extended in the future in order to support more than one instruction preceding a return instruction.

To scan the codebase for useful instruction sequences, the Constructor first enumerates all sections of the PE file that contain executable code and scans these for x86 `ret` opcodes. In addition to the standard `ret` instruction, which has the opcode `0xC3`, we are also interested in return instructions that add an immediate value to the stack, represented by opcode `0xC2` and followed by the 16bit immediate offset. The former are favorable to the latter as they induce less memory consumption in the stack allocation since we need to append effectively unused memory before the next instruction.

Having found all available return instructions, the Constructor then bytewise disassembles the sequence backwards, thereby building a trie. This works analogoulsy to the method already described by Shacham [23]. In order to disassemble encoded x86 instructions, our program uses the *distorm* library [10].

3.1.2 Building Gadgets

The next logical step is chaining together instruction sequences to form structured gadgets that perform basic operations. Gadgets built by the Constructor form the very basic entities that are chained together by the Compiler for building the program stack allocation. Due to the clear separation of the Constructor and the Compiler, final gadgets are independent of each other. Therefore, each final gadget constitutes an autonomous piece of return-oriented code: Final gadgets take a set of source operands, perform a well-defined operation on these, and then write the result into a destination operand. In our model, source and destination operands are always memory variable addresses. For example, an addition gadget takes two source operands, i.e., memory addresses to both input variables, as input, adds both values, and then writes back the result to the memory address pointed to by the destination operand. There are certain exceptions to this rule, namely final gadgets that perform very specific tasks for certain situations, e.g., manipulating the stack register directly. Final gadgets are designed to be fine-grained with respect to the constraints imposed by the operand model. They can be separated into three classes: Arithmetic, logical and bitwise operations; control flow manipulations (static and dynamic); and stack register manipulations.

The crucial point in gadget construction concerns the algorithm that is deployed to spot appropriate useful instructions and the rules in which they are chained together. We consider completeness, memory consumption, and runtime to be the three dominating properties. By completeness, we mean the algorithm's fundamental ability to construct gadgets even in a minimal codebase,

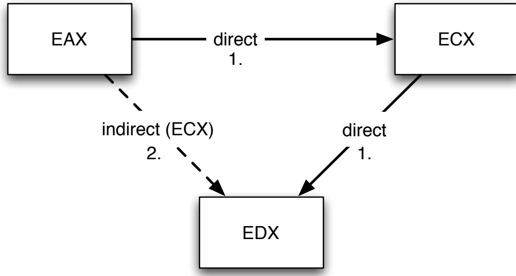

Figure 3: MOV connection graph: Chained instructions can be used to emulate other instructions.

where minimal indicates a codebase with a theoretically minimal set of instruction sequences to allow for corresponding gadget computations. By memory consumption, we denote that the constructed gadgets should be preferably small in size. By runtime, we mean that the algorithm should terminate within a reasonable period of time. Due to the CISC nature of x86 and the corresponding complexity of the machine language, we consider the completeness property to be the most difficult one to achieve. The many subtle details of this platform make it hard to find all possible combinations of useful instruction performing a given operation.

In the following, we provide a deeper look into our gadget construction algorithm. As with every modern CPU, x86 is a register-based architecture. This observations drives the starting point of our algorithm in that its first step is to define a set of general purpose registers that are allowed to be used, i.e., read from or written to, by gadget computations. This also has the positive side-effect that it enables an easy way to control which registers are modified by the return-oriented program. We will henceforth call these registers *working registers*.

Basic Gadgets. Starting from this point, we gradually construct lists of gadgets performing a related task for each working register. More precisely, the first step is to check which register can be loaded with fixed values, an operation that can easily be achieved with a `pop <register>; ret` sequence (*register-based constant load gadgets*). Afterwards, the Constructor searches for unary instructions sequences, e.g., `not` or `neg`, that take working registers as their operands (*register-based unary operation gadgets*). Subsequently, the algorithm checks which working registers are connected by binary instruction sequences, e.g., `mov`, `add`, `and`, and the like (*register-based binary operation gadgets*). In order to find indirectly connected registers, we build a directed graph for each operation whereas a node represents a register and an edge depicts an operation from the source register to the destination register (also always being a source operand on x86). Then, we traverse all paths in the graph for each node. For example, let us assume the following situation: The given codebase allows for the execution of `mov ecx, eax; ret` and `mov edx, ecx; ret` sequences, but does not supply `mov edx, eax` sequences. We can easily find the corresponding path in our graph and hence construct a gadget that moves the content of `eax` to `edx` by chaining together both sequences (see Fig. 3). Since x86 is not a load-store-architecure, i.e., most instructions may take direct memory operands, we also search for memory operand based instructions (*register-based memory load-/operation gadgets*). This also allows us to check which working registers can be loaded with memory contents, for instance, `mov eax, [ecx]; ret` easily allows us to load an arbitrary memory location into `eax` by preparing `ecx` accordingly. The result of this first stage of the algorithm are lists of internal gadgets being bound to working registers and performing certain operations on these.

In the next stage, our algorithm merges working register-based gadgets to form new, final gadgets that perform certain operations, e.g., addition, multiplication, bitwise OR, and so on (*final unary/binary operation gadgets*). Therefore, it generates every possible combination of according register-based load/store and operation gadgets to choose the one being minimal with respect to consumed memory space. In the construction, we have to take into account possibly emerging side-effects when connecting instruction sequences. We say that a gadget has a *side-effect* on a given register when it is modified during execution. For instance, if we wish to build a gadget that loads two memory addresses into `eax` and `ecx` and appends an `and eax, ecx; ret` sequence, we have to make sure that both load gadgets do not have side-effects on each other's working register.

Control Flow Alteration Gadgets. Afterwards, the algorithm constructs final gadgets that allow for static and dynamic control flow alterations in a return-oriented program (*final comparison and dynamic control flow gadgets*). Therefore, we must first compare two operands with either a `cmp` or `sub` instruction, both have the same impact on the `eflags` registers which holds the condition flags. The main problem in this context is gaining access to the CPU's flag registers as this is only possible with a limited set of instructions. As already pointed out by Shacham [23], a straightforward solution is to search for the `lahf` instruction, which stores the lower part of the `eflags` register into `ah`. Another possibility is to search for `setCC` instructions, which store either one or zero depending on whether the condition is true or not. Thereby, `CC` can be any condition known to the CPU, e.g., equal, less than, greater or equal, and so on. Once

we have stored the result of the comparison (where 1 means true and 0 means false) the natural way to proceed is to multiply this value by four and add it to a jump table pointer. Then, we simply move the stack register to the value being pointed at.

Additional Gadgets. Finally, the Constructor builds some special gadgets that enable very specific tasks, such as, e.g., pointer dereferencing (*final dereferencing gadgets*), and direct stack or base register manipulation (*stack register manipulation gadgets*). The latter are required in certain situation as described in the next section. The final output of the Constructor is an XML file that describes the finals gadgets along with a list of exported symbols from the codebase.

Turing Completeness. Gadgets are used within return-oriented programming as the basis blocks of each computation. An interesting question is now which kind of gadgets are needed such that return-oriented programming is *Turing complete*, i.e., it can compute every Turing-computable function [30]. We construct gadgets to load/store variables (including pointer dereferencing), branch instructions, and also gadgets for arithmetic operations (i.e., addition and not). This set of gadgets is minimal in the sense that we can construct from these gadgets any program: Our return-oriented framework can implement so called *GOTO languages*, which are Turing complete [12].

3.2 Compiler

The Compiler is the next building block of our return-oriented framework: This tool takes the final gadgets constructed by the Constructor along with a high-level language source file as input to produce the stack allocation for the return-oriented program. The Compiler acts as an abstraction of the concrete codebase so that developers do not have to mess with the intricacies of the codebase on the lowest layer; moreover, it provides a comparatively easy and abstract way to formulate a complex task to be realized in a return-oriented fashion. The Compiler's output describes the stack allocation as well as additional memory areas serving a specific purpose in a position independent way, i.e., it only contains relative memory addresses. This stems from the fact that the Compiler cannot be aware of the final code locations since drivers may be relocated in kernel memory due to address conflicts. Moreover, the program memory's base location may be unknown at this stage. It is hence the task of the Loader to resolve these relative addresses to absolute addresses (see next section).

3.2.1 Dedicated Language

Naturally, one of the first considerations in compiler development concerns the programming language employed. One possibility is to build the Compiler on top of an already existing language, ideally one that is designed the accomplish low-level tasks, such as C. However, this also introduces a profound overhead as all the language's peculiarities, e.g., the entire type system, must be implemented in a correct manner. Due to our very specific needs, we have found none of the existing language to be suited for our purpose and thus decided to create a dedicated language. It bears certain resemblance to many existing languages, specifically C. Our dedicated language provides the following code constructs:

- subroutines and recursive subroutine calls,
- a basic type-system that consists of two variable types,
- all arithmetic bitwise, logical and pointer operators known to the C language with some minor deviations, and
- nested dynamic control flow decisions and nested conditional looping.

Additionally, we also supports external subroutine calls which enables one to dispatch operations to exported symbols from drivers or the kernel; this gives us more flexibility, greatly simplifies the development of return-oriented programs, and also substantially decreases stack allocation memory consumption.

Two basic variable types are supported: Integers and character arrays, the former being 32bit long while in case of the latter, strings are zero-terminated just as in C. Along with the ability to call external subroutines, this enables us to use standard C library functions exported by the kernel to process strings within the program. We do not need support for short and char integers for now as we do not consider these to be substantially relevant for our needs. Short integer operations thus must be emulated by the return-oriented program when needed.

The Compiler has been implemented in C++ using the ANTLR compiler generation framework [29]. Source code examples for our dedicated programming language are introduced in Section 5.4 and in Appendix B.

3.2.2 Memory Layout

Just as the Constructor chains together instruction sequences, the Compiler chains together gadgets to build a program performing the semantics imposed by the source code. Apart from that, it also defines the memory layout and assigns code and data to memory locations. By *code*, we henceforth mean the stack allocation of the

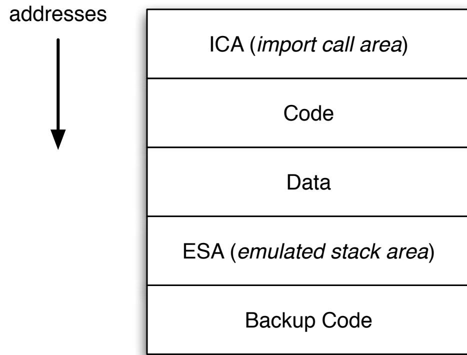

Figure 4: Memory layout of program image within our return-oriented framework

sum of all gadgets of a program (mostly return addresses to instruction sequences); this must not be confused with real CPU code, i.e., the code as we defined does not need *any* executable memory, but appears like usual data to the processor. This is the key concept in bypassing kernel integrity protection mechanisms: We do not need to inject code since we re-use existing code within the kernel during an exploit.

When we use the term *data*, we henceforth mean the memory area composed by the program's variables and temporary internal data required by computations. We then constitute the memory layout to consist of a linear memory space we hereafter call the *program image*, which is shown in Fig. 4. Furthermore, some regions of this space serve special purposes we describe later on. In total, we separate the program image into five sections: Code, data, ICA, ESA and backup code.

The so-called *import call area* (ICA) resides at the very beginning, i.e., the lowest address, of the address space. When executing external function calls, the program prepares the call to be dispatched with the stack pointer `esp` pointing at the end (the highest address) of the ICA. Therefore, it first prepares this region by copying the arguments and return addresses to point to specific stack manipulation gadgets. Special care has to be taken concerning the imposed calling convention of the callee. We support both relevant conventions, namely `stdcall`, i.e., the callee cleans up the stack, and `cdecl`, i.e., the caller cleans up the stack. The need for such a dedicated section stems from the fact that, upon entry, the callee considers all memory regions below `esp` to be available to hold its local variables, hence overwriting return-oriented code that might still be needed at a later stage, i.e., when a jump back occurs.

Following the ICA, the Compiler places the code, i.e., return addresses and constant values to be popped into registers, followed by the data section which holds the actual explicit variables as well as some implicit temporary variables that are mandatory during computation. After that, the *emulated stack area* (ESA) resides, which is used to emulate a "stack in the stack" to allow for recursive subroutine calls in the return-oriented program. The program image is terminated by an optional backup of the code section, a necessity that arises from a peculiarity of the Windows operating system we discuss later on in Section 5.2.

3.2.3 Miscellaneous

We also provide special language constructs enabling one to retransfer the CPU control flow to a non-return oriented source. For instance, in the typical case of an exploit and subsequent execution of return-oriented code, we might wish to return to the vulnerable code to allow for a continuation of execution. Therefore, we must restore the `esp` register to point to its original value. Our language hence provides appropriate primitives to tamper with the stack.

3.3 Loader

The final building block of our system consists of the Loader whose main task is to resolve the program image's relative addresses to absolute addresses. Therefore, it must first enumerate all loaded drivers in the system and retrieve their base addresses. Luckily, Windows provides a function by the name of `EnumDeviceDrivers` that lets us accomplish this task even in usermode.

For the sake of flexibility, the Loader is implemented as a dynamic link library (DLL). The actual exploit transfers the task of building the final program image to the Loader and then adjusts the exploit to modify the instruction pointer `eip` to a gadget that modifies the stack (e.g., `pop esp; ret`) to start the execution of the return-oriented program. It is therefore sufficient for the exploit to be able to modify eight subsequent bytes in the stack frame: The first four bytes are a return address (of the sequence `pop esp; ret`) that is executed upon the next `ret` in the current control flow; the last four bytes point to the entry point of the program image to which control will flow after the execution of the next `ret`.

4 Evaluation Results

We implemented the system we described in the previous section in the C++ programming language. The Constructor consists of about 3,400 lines of code (LOC), whereas the Compiler is implemented in about 3,200 LOC. The loader only needs 700 LOC.

In the following, we present evaluation results for the individual components of our framework. We first show measurement results for the Constructor and Compiler and then provide several examples of the gadgets constructed by our tools. Finally we also measure the runtime overhead of return-oriented programs.

4.1 Constructor and Compiler

4.1.1 Evaluation of Useful Instructions and Gadget Construction

One goal of our work is to fully automate the process of constructing gadgets from kernel code on different platforms without the need of manual analysis of machine language code. We thus tested the Constructor on ten different machines running different versions of Windows as operating system: Windows 2003 Server, Windows XP, and Windows Vista were considered in different service pack versions to assess a wide variety of platforms. On each tested platform the Constructor was able to find enough useful instructions to construct all important gadgets that are needed by the Compiler, i.e., on each platform we are able to compile arbitrary return-oriented programs. This substantiates our claim that our framework is general and portable.

Table 1 provides an overview of the results for the gadget construction algorithm for six of the ten test configurations. We omitted the remaining four test results for the sake of brevity; the results for these machines are very similar to the listed ones. The table contains test results for two scenarios: On the one hand, we list the number of return instructions and trie leaves when using *any* kernel code, e.g., all drivers and kernel components. On the other hand, we list in the restricted column (res.) the results when using *only* the main kernel component (`ntoskrnl.exe`) and the Win32-subsystem (`win32k.sys`) for extracting useful instructions. These two components are available in any Windows environment and thus constitute a memory region an attacker can always use to build gadgets.

The number of return instructions found varies with the platform and is influenced by many factors, mainly OS version/service pack and hardware configuration. Especially the hardware configuration can significantly enlarge the number of available return instructions since the installed drivers add a large codebase to the system: We found that often graphic card drivers add thousands of instructions that can be used by an attacker. For the complete codebase we found that on average every 162nd instruction is a return. Therefore an attacker typically finds tens of thousands of instructions she can use.

If the attacker restricts herself to using only the core kernel components, she is still able to find enough return instructions to be able to construct all necessary gadgets: We found that on average every 153rd instruction is a return, indicating a more dense structure within the core kernel components. These returns and the preceding instructions could be used to construct the gadgets in all tested environments. This result indicates that on Window-based systems an attacker can implement an arbitrary return-oriented program since all important gadgets can be built.

The most common instruction preceding a return is `pop ebp`: On average across all tested systems, this instruction was found in about 72% of the cases. This is no surprise since the sequence `pop ebp; ret` is the standard exit sequence for C code. Other common instructions the Constructor finds are `add esp, <const>` (12.2%), `pop (eax|ecx|edx)` (4.2%), and `xor eax, eax` (3.7%). Other instructions can be found rather seldom, but if a given instruction occurs at least once the attacker can use it. For example, the instruction `lahf`, which is used to access the CPU's flag registers, was commonly found less than 10 times, but nevertheless the attacker can take advantage of it.

4.1.2 Gadget Examples

In order to illustrate the gadgets constructed by our framework, we present a few examples of gadgets in this section. A full listing of all gadgets constructed during the evaluation on ten different machines is available online [13] such that our results can be verified.

Figure 5 shows the AND gadget constructed on two different machines both running Windows XP SP2. In each of the sub-figures, the left part displays the instructions that are actually used for the computation: Remember that our current implementation considers one instruction preceding a return instruction, i.e., after each of the displayed instructions one implicit `ret` instruction is executed. The right part shows the memory locations where the instruction is found within kernel memory (R), or indicates the label name (L). Labels are memory variable addresses.

The two gadgets each perform a logical AND of two values. This is achieved by loading the two operands into the appropriate registers (`pop, mov` sequence), then performing the `and` instruction on the registers, and finally writing the result back to the destination address. Although both programs are executed on Windows XP SP2 machines, the resulting return-oriented code looks completely different since useful instructions in different kernel components are used by the Constructor.

Another example of a gadget constructed by our framework is shown in Figure 6. The left example shows a gadget for a machine running Windows Vista, while the example on the right hand side is constructed on a ma-

Machine configuration	# ret inst.	# trie leaves	# ret inst. (res)	# trie leaves (res)
Native / XP SP2	118,154	148,916	22,398	25,968
Native / XP SP3	95,809	119,533	22,076	25,768
VMware / XP SP3	58,933	67,837	22,076	25,768
VMware / 2003 Server SP2	61,080	70,957	23,181	26,399
Native / Vista SP1	181,138	234,685	30,922	36,308
Bootcamp / Vista SP1	177,778	225,551	30,922	36,308

Table 1: Overview of return instructions found and generated trie leaves on different machines

```
pop ecx             | R: ntkrnlpa.exe:0006373C
                    | L: <RightSourceAddress>+4
mov edx, [ecx-0x4]  | R: vmx_fb.dll:00017CBD
pop eax             | R: ntkrnlpa.exe:000436AE
                    | L: <LeftSourceAddress>
mov eax, [eax]      | R: win32k.sys:000065D1
and eax, edx        | R: win32k.sys:000ADAE6
pop ecx             | R: ntkrnlpa.exe:0006373C
                    | L: <DestinationAddress>
mov [ecx], eax      | R: win32k.sys:0000F0AC
```

```
pop ecx             | R: nv4_mini.sys:00005A15
                    | L: <RightSourceAddress>-4
pop eax             | R: nv4_mini.sys:00074EF2
                    | L: <LeftSourceAddress>
mov eax, [eax]      | R: nv4_disp.dll:00125F30
and eax, [ecx+0x4]  | R: sthda.sys:000024ED
pop ecx             | R: nv4_mini.sys:00005A15
                    | L: <DestinationAddress>
mov [ecx], eax      | R: nv4_disp.dll:000DE9DA
```

Figure 5: Example of two AND gadgets constructed on different machines running Windows XP SP2. The implicit ret instruction after each instruction is omitted for the sake of brevity.

chine running Windows 2003 Server. Again, the memory locations of the gadget instructions are completely different since the Constructor found different useful instruction sequences that are then used to build the gadget.

4.2 Runtime Overhead

The average runtime of the Constructor for the restricted set of drivers that should be analyzed is 2,009 ms, thus the time for finding and constructing the final gadgets is rather small.

To assess the overhead of return-oriented programming in real-world settings, we also measured the overhead of an example program written within our framework compared to a "native" implementation in C. Therefore, we implemented two identical versions of QuickSort, one in C and one in our dedicated return-oriented language. The source code of the latter can be seen in Appendix B.

Both algorithms sort an integer array of 500,000 randomly selected elements and the evaluations were carried out on an Intel Core 2 Duo T7500 based notebook running Windows XP SP3. The C code was compiled with Microsoft Visual Studio 2008; in order to improve the fundamental expressiveness of the comparison, all compiler optimizations were disabled. Each algorithm was executed three times and we calculated the average of the runtimes.

The return-oriented QuickSort took *21,752 ms* on average compared to *161 ms* for C QuickSort. The results clearly show that the overhead imposed by return-oriented programs is significant; on average, they were 135 times slower than their C counterparts. We would like to stress that we did not build our system with speed optimizations in mind. Additionally, in our domain, return-oriented rootkits usually do not involve time-intensive computations, thus the slowness might not be a problem in practice. On the other hand, the overhead might well be exploited by detection mechanisms that try to find return-oriented programs.

5 Return-Oriented Rootkit

In order to evaluate our system in the presence of a kernel vulnerability, we have implemented a dedicated driver containing insecure code. Remember that our attack model includes this situation. By this example, we show that our systems allows us to implement a return-oriented rootkit in an efficient and portable manner. This rootkits bypasses kernel code integrity mechanisms like NICKLE and SecVisor since we do not inject new code into the kernel, but only execute code that is already available. While the authors of NICKLE and SecVisor acknowledge that such a vulnerability could exist [19, 22], we are the first to actually show an implementation of an attack against these systems. In the following, we first introduce the different stages of the infection process and afterwards describe the internals of our rootkit example.

```
'LoadEspPointer' gadget:
pop ecx         | R: nvlddmkm.sys:000156F5
                | L: <Address>
mov eax, [ecx] | R: ntkrnlpa.exe:002D15C3
mov eax, [eax] | R: win32k.sys:000011AE
pop ecx         | R: nvlddmkm.sys:000156F5
                | L: &<LocalVar>
mov [ecx], eax | R: ntkrnlpa.exe:0002039B
pop esp         | R: nvlddmkm.sys:00036A54
                | L: <LocalVar>
```

```
'LoadEspPointer' gadget:
pop eax         | R: ntkrnlpa.exe:0001CD4F
                | L: <Address>
mov eax, [eax] | R: win32k.sys:00087E17
mov eax, [eax] | R: win32k.sys:00087E17
pop ecx         | R: ntkrnlpa.exe:00080A8D
                | L: &<LocalVar>
mov [ecx], eax | R: win32k.sys:000A8DDB
pop esp         | R: ntkrnlpa.exe:00081A67
                | L: <LocalVar>
```

Figure 6: Example of gadget constructed on a machine running Windows Vista SP1 (left) and Windows 2003 Server (right). Again, the implicit `ret` instruction after each instruction is omitted.

5.1 Experimental Setup

Vulnerability. As already stated, we assume the presence of a vulnerability in kernel code that enables an exploit to corrupt the program flow in kernel mode. More precisely, our dedicated driver contains a specially crafted buffer overflow vulnerability that allows an attacker to tamper with the kernel stack. The usual way to implement driver-to-process communication is to provide a device file name being accessible from userspace. The process hence opens this device file and may send data to the driver by writing to it. Write requests trigger so-called *I/O request packets* (IRP) at the driver's callback routine. The driver then takes the input data from userspace and copies it into its own local buffer without validating its length. This leads to a classical buffer overflow attack and enables us to write stack values of arbitrary length.

Exploit. We exploit this vulnerability by writing an oversized buffer to the device file, thereby replacing the return value on the stack to point to a `pop esp; ret` sequence, and the next stack value to point to the entrypoint of the return-oriented program. By overwriting these eight bytes, we manage to modify the stack register to point to the beginning of our return-oriented program. Of course, the vulnerability itself may vary in its concrete nature, however, any similar insecure code allows us to mount our attack: A single vulnerability within the kernel or any third-party driver is enough to attack a system and start the return-oriented program.

The only question that remains is where to put the program image. We basically have two options: First, the exploit could overwrite the entire kernel stack with our return-oriented program; in case of the above vulnerability, this would be possible as there is no upper limit. In case of Windows, the kernel stack size has a fixed limit of 3 pages which heavily constrains this option. Second, the exploit could, at least initially, keep the program image in userspace memory. We prefer the latter approach to implement our *rootkit loader*, although it has some implications that need to be addressed as we now explain.

5.2 Intricacies in Practice

One of the main practical obstacles that we faced stems from the way how Windows treats its kernel stack. All current Windows operating systems separate kernel space execution into several *interrupt request levels* (IRQL). IRQLs introduce a priority mechanism into kernel-level execution and are similar to user-level thread priorities. Every interrupt is executed in a well-defined IRQL; whenever such an interrupt occurs, it is compared to the IRQL of the currently executing thread. In case the current IRQL is above the requested one, the interrupt is queued for later rescheduling. As a consequence, an interrupt cannot suspend a computation running at a higher IRQL. This has some implications concerning accessing pageable memory in kernel mode since page-access exceptions are being processed in a specific IRQL (`APC_LEVEL`, to be precise) while other interrupts are handled at higher IRQLs. Hence, the kernel and drivers must not access pageable memory areas at certain IRQLs.

Unfortunately, this leads to some problems due to a peculiarity of Windows kernels: Whenever interrupts occur and hence must be handled, the Windows kernel borrows the current kernel stack to perform its interrupt handling. Therefore, the interrupt handler allocates the memory *below* the current value of `esp` as the handler's stack frame. While this is totally acceptable in common situations, it leads to undesirable implications in case of return-oriented programs as the stack values below the current stack pointer may indeed be needed in the subsequent execution. As described in Section 3.1.2, control flow branches are stack register modifications: When the program wants to jump backwards, it may fail at this point since the prior code might have been overwritten by the interrupt handler in the first place. To solve this problem, the Compiler provides an option to dynamically restore affected code areas: Whenever a return-oriented

control flow transition backwards occurs (which hence could have been subject to unsolicited modifications), we first prepare the ICA to perform a `memcpy` call that restores the affected code from the backup code section and subsequently performs the return-oriented jump. This works since the ICA is located below the code section and hence the code section cannot be overwritten during the call. The data and backup section will never be overwritten as they are always on top of every possible value of `esp`.

Furthermore, we will also run into IRQL problems in case the program stack is located in pageable memory: As soon as an interrupt is dispatched above `APC_LEVEL`, a blue-screen-of-death occurs. This problem should be overcome by means of the `VirtualLock` function which allows a process to lock a certain amount of pages into physical memory, thereby eliminating the possibility of paging errors on access. However, due to reasons which are yet not known to us, this does not always work as intended for memory areas larger than one page. We have frequently encountered paging errors in kernelmode although the related memory pages have previously been locked. We therefore introduce a workaround for this issue in the next section.

5.3 Rootkit Loader

To overcome the paging IRQL problem, we have implemented a pre-step in the loading phase. More precisely, in the first stage, we prepare a tiny return-oriented rootkit loader that fits into one memory page and prepares the entry of the actual return-oriented rootkit. It allocates memory from the kernel's non-paged pool, which is definitely never paged out, and copies the rootkit code from userspace before performing a transition to the actual rootkit. This has proven to work reliably in practice and we have not encountered any further IRQL problems. Again, the Rootkit Loader program image resides in userspace, which limits the ability of kernel integrity protection mechanisms to prohibit the loading of our rootkit.

5.4 Rootkit Implementation

To demonstrate our system's capability, we have implemented a return-oriented rootkit that is able to hide certain system processes. This is achieved by an approach similar to the one introduced by Hoglund and Butler [11]: Our rootkit cycles through Windows' internal process block list to search for the process that should be hidden and, if successful, then modifies the pointers in the doubly-linked list accordingly to release the process' block from the list. Since the operating system holds a separate, independent scheduling list, the process will

```
int ListStartOffset =
  &CurrentProcess->process_list.Flink -
  CurrentProcess;
int ListStart =
  &CurrentProcess->process_list.Flink;
int ListCurrent = *ListStart;
while(ListCurrent != ListStart) {
  struct EPROCESS *NextProcess =
    ListCurrent - ListStartOffset;
  if(RtlCompareMemory(NextProcess->ImageName,
                      "Ghost.exe", 9) == 9) {
    break;
  }
  ListCurrent = *ListCurrent;
}

if(ListCurrent != ListStart) {
  // process found, do some pointer magic
  struct EPROCESS *GhostProcess =
    ListCurrent - ListStartOffset;

  // Current->Blink->Flink = Current->Flink
  GhostProcess->process_list.Blink->Flink =
    GhostProcess->process_list.Flink;

  // Current->Flink->Blink = Current->Blink
  GhostProcess->process_list.Flink->Blink =
    GhostProcess->process_list.Blink;

  // Current->Flink = Current->Blink = Current
  GhostProcess->process_list.Flink =
    ListCurrent;
  GhostProcess->process_list.Blink =
    ListCurrent;
}
```

Figure 7: Rootkit source code snippet in dedicated language for return-oriented programming that can be compiled with our Compiler.

still be running in the system, albeit not being present in the results of process enumeration requests: The process is hidden within Windows and not visible within the Taskmanager. Figure 8 in Appendix A illustrates the rootkit in practice.

Figure 7 shows an excerpt of the rootkit source code written in our dedicated language. This snippet shows the code for (a) finding the process to be hidden and (b) hiding the process as explained above.

Once the process hiding is finished, the rootkit performs a transition back to the vulnerable code to continue normal execution. This seems to be complicated since we have modified the stack pointer in the first place and must hence restore its original value. However, in practice this turns out to be not problematic since this value is available in the thread environment block that is always located at a fixed memory location. Hence, we reconstruct the stack and jump back to our vulnerable driver. Besides process hiding, arbitrary data-driven attacks can be implemented in the same way: The rootkit needs to

exploit the vulnerability repeatedly in order to gain control and can then execute arbitrary return-oriented programs that perform the desired operation [3].

We would like to mention at this point that more sophisticated rootkit functionality, e.g., file and socket hiding, might demand more powerful constructs, namely *persistent* return-oriented callback routines. Data-only modifications as implemented by our current version of the rootkit hence might not be sufficient in this case. In contrast to Riley et al. [19], we do believe that this is possible in the given environment by the use of specific instruction sequences. However, we have not yet had the time to prove our hypothesis and hence leave this topic up to future work in this area.

The rootkit example works on Windows 2000, Windows Server 2003 and Windows XP (including all service packs). We did not port it to the Vista platform yet as the publicly available information on the Vista kernel is still limited. We also expect problems with the Vista *PatchGuard*, a kernel patch protection system developed by Microsoft to protect x64 editions of Windows against malicious patching of the kernel. However, we would like to stress that PatchGuard runs at the same privilege level as our rootkit and hence could be defeated. In the past, detailed reports showed how to circumvent Vista PatchGuard in different ways [24, 26, 25].

6 Conclusion and Future Work

In this paper we presented the design and implementation of a framework to automate return-oriented programming on commodity operating systems. This system is portable in the sense that the Constructor first enumerates what instruction sequences can be used and then dynamically generates gadgets that perform higher-level operations. The final gadgets are then used by the Compiler to translate the source code of our dedicated programming language into a return-oriented program. The language we implemented resembles the syntax of the C programming language which greatly simplifies developing programs within our framework. We confirmed the portability and universality of our framework by testing the framework on ten different machines, providing deeper insights into the mechanisms and constraints of return-oriented programming. Finally we demonstrated how a return-oriented rootkit can be implemented that circumvents kernel integrity protection systems like NICKLE and SecVisor.

In the future, we want to investigate effective detection techniques for return-oriented rootkits. We also plan to extend the research in two other important directions. First, we plan to examine how the current rootkit can be improved to also support *persistent* kernel modifications. This change is necessary to implement rootkit functions like hiding of files or network connections, which require a persistent return-oriented callback routine. This change would enhance the rootkit beyond the current data-driven attacks. Second, we plan to analyze how the techniques presented in this paper could be used to attack control-flow integrity [1, 7, 14, 18] or data-flow integrity [5] mechanisms. These mechanisms are orthogonal to the kernel integrity protection mechanisms we covered in this paper.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity – Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[2] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, October 1972. AD-758 206, ESD/AFSC.

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

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

[5] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, May 2008.

[7] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[8] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report Cylab Technical Report CMU-CyLab-08-008, CMU, June 2008.

[9] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS)*, February 2003.

[10] Gil Dabah. diStorm64 - The ultimate disassembler library. `http://www.ragestorm.net/distorm`, 2009.

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

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[13] Ralf Hund. Listing of gadgets constructed on ten evaluation machines. `http://pi1.informatik.uni-mannheim.de/filepool/projects/return-oriented-rootkit/measurements-ro.tgz`, May 2009.

[14] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[15] Microsoft. Digital Signatures for Kernel Modules on Systems Running Windows Vista. `http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kmsigning.doc`, July 2007.

[16] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. `http://support.microsoft.com/kb/875352`, 2008.

[17] Nergal. The advanced return-into-lib(c) exploits: PaX case study. `http://www.phrack.org/issues.html?issue=58&id=4`, 2001.

[18] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, October 2007.

[19] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[20] Ryan Riley, Xuxian Jiang, and Dongyan Xu. NICKLE: No Instructions Creeping into Kernel Level Executed. `http://friends.cs.purdue.edu/dokuwiki/doku.php?id=nickle`, 2008.

[21] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Techniques. `http://www.suse.de/~krahmer/no-nx.pdf`, September 2005.

[22] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[23] 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)*, October 2007.

[24] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, 3, January 2006.

[25] Skywing. PatchGuard Reloaded: A Brief Analysis of PatchGuard 3. *Uninformed*, 8, September 2007.

[26] Skywing. Subverting PatchGuard Version 2. *Uninformed*, 6, January 2007.

[27] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/0063.html`, 1997.

[28] PaX Team. Documentation for the PaX project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

[29] Terence Parr. ANTLR Parser Generator. `http://www.antlr.org`, 2009.

[30] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

A Return-Oriented Rootkit in Practice

Figure 8 depicts the results of an attack using our return-oriented rootkit: The process Ghost.exe (lower left window) is a simple application that periodically prints a status message on the screen. The rootkit (upper left window) first exploits the vulnerability in the driver to start

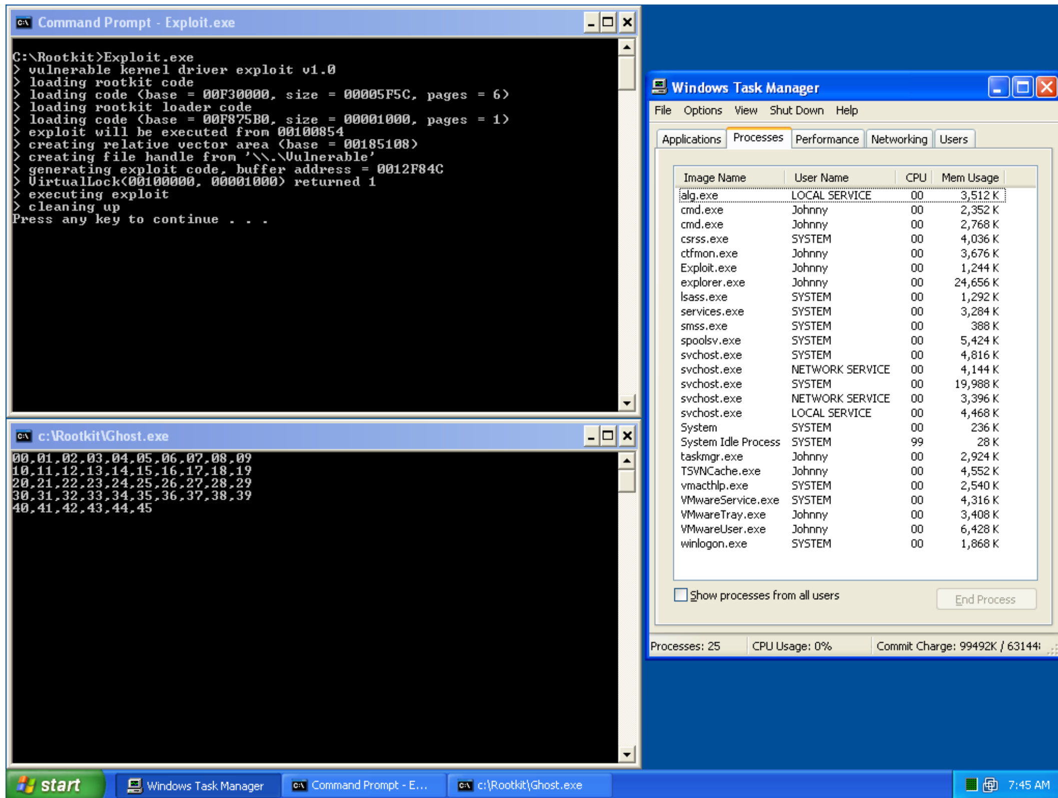

Figure 8: Return-oriented rootkit in practice, hiding the process Ghost.exe.

the return-oriented program. This program then hides the presence of Ghost.exe as explained in Section 5.4: The process Ghost.exe is running, however, the rootkit removed it from the list of running processes and it is not visible in the Taskmanager.

B Return-Oriented QuickSort

The following listing shows an implementation of Quick-Sort within our dedicated programming language. The syntax is close to the C programming language, allowing a programmer to implement a return-oriented program without too much effort. The most notable exception from C's syntax is related to importing of external functions: Our language can import subroutine calls from other libraries, enabling an easy way to call external functions like `printf()` or `srand()`. However, each function needs to be imported explicitly. Furthermore, the language implements only a basic type-system consisting of integers and character arrays, but this should not pose a limitation.

```
import("msvcrt.dll", printf:cdecl,
                     srand:cdecl,
                     rand:cdecl,
                     malloc:cdecl);

import("kernel32.dll", GetCurrentProcess,
                       TerminateProcess,
                       GetTickCount);

int data;
int size = 500000;

function partition(int left, int right,
                   int pivot_index) {
  int pivot = data[pivot_index];
  int temp = data[pivot_index];
  data[pivot_index] = data[right];
  data[right] = temp;
  int store_index = left;
  int i = left;

  while(i < right) {
    if(data[i] <= pivot) {
      temp = data[i];
      data[i] = data[store_index];
      data[store_index] = temp;
      store_index = store_index + 1;
    }
    i = i + 1;
  }

  temp = data[store_index];
  data[store_index] = data[right];
  data[right] = temp;

  return store_index;
}
```

```
function quicksort(int left, int right) {
  if(left < right) {
    int pivot_index = left;
    pivot_index = partition(left, right,
                            pivot_index);
    quicksort(left, pivot_index - 1);
    quicksort(pivot_index + 1, right);
  }
}

function start() {
  printf("Welcome to ro-QuickSort\n");
  data = malloc(4 * size);
  srand(GetTickCount());
  int i = 0;
  while(i < size) {
    data[i] = rand();
    i = i + 1;
  }

  int time_start = GetTickCount();
  quicksort(0, size - 1);
  int time_end = GetTickCount();
  printf("Sorting completed in %u ms:\n",
         time_end - time_start);
}
```