UNIVERSITY OF CALIFORNIA, SAN DIEGO

Finding the Bad in Good Code:
Automated Return-Oriented Programming Exploit Discovery

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Ryan Glenn Roemer

Committee in charge:

Professor Stefan Savage, Chair
Professor Hovav Shacham, Co-Chair
Professor Geoffrey Voelker

2009

The thesis of Ryan Glenn Roemer is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____
Co-Chair

_____
Chair

University of California, San Diego

2009

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

List of Tables vii

Acknowledgements viii

Abstract of the Thesis x

Chapter 1 Introduction 1

Chapter 2 Evolution: From Stack-Smashing to Return-Oriented Programming 5
2.1 Traditional Stack-Smashing and Defenses 5
2.2 W⊕X and Return-to-Libc 6
2.3 Return-Oriented Programming on x86 7

Chapter 3 SPARC Architecture Overview 9
3.1 Registers 9
3.2 Register Banks 10
3.3 The Stack and Subroutine Calls 11
3.4 Buffer Overflows and Return-to-Libc 11

Chapter 4 Return-Oriented Programming on SPARC 14
4.1 Finding SPARC Instruction Sequences in libc 16
4.2 Constructing SPARC Gadgets 17
4.3 Crafting a Return-Oriented Program 17
4.4 Gadget Abstractions and Practical Return-Oriented Programming 18

Chapter 5 SPARC Gadget Catalog 20
5.1 Memory 21
5.1.1 Address Assignment 21
5.1.2 Pointer Read 21
5.1.3 Pointer Write 22
5.2 Assignment 22
5.2.1 Constant Assignment 22
5.2.2 Variable Assignment 23
5.3 Arithmetic 24

5.3.1 Increment, Decrement . . . 24
5.3.2 Addition, Subtraction, Negation . . . 25
5.4 Logic . . . 26
5.4.1 And, Or, Not . . . 26
5.4.2 Shift Left, Shift Right . . . 27
5.5 Control Flow . . . 27
5.5.1 Branch Always . . . 29
5.5.2 Branch Equal; Branch Less Than or Equal; Branch Greater Than . . . 29
5.5.3 Branch Not Equal; Branch Less Than; Branch Greater Than or Equal . . . 30
5.6 Function Calls . . . 32
5.7 System Calls . . . 34

Chapter 6 Automated Gadget Searching . . . 36
6.1 Gadget Search Tool . . . 37
6.2 Search Queries . . . 38
6.2.1 Arguments . . . 38
6.2.2 Instructions . . . 42
6.2.3 Sequences, Gadgets, and Gadget Lists . . . 45
6.3 Search Algorithm . . . 46
6.3.1 Gadget Matching . . . 46
6.3.2 Instruction Sequence Matching . . . 47

Chapter 7 Vulnerability Analysis . . . 50
7.1 Measurement Methodology . . . 50
7.2 First Binary Group, Untrained . . . 52
7.2.1 Results . . . 53
7.3 First Binary Group, Trained . . . 57
7.3.1 Methodology . . . 58
7.3.2 Query Modifications and Training . . . 58
7.3.3 Results . . . 61
7.3.4 Results Comparison . . . 64
7.4 Second Binary Group . . . 66
7.4.1 Results . . . 66
7.5 Lessons and Implications for Automated Gadget Searching 69

Chapter 8 Conclusion . . . 71

Bibliography . . . 73

LIST OF FIGURES

Figure 3.1: SPARC Stack Layout 12

Figure 4.1: Return-Oriented Program 15

Figure 5.1: Pointer Read (`v1 = *v2`) 21
Figure 5.2: Pointer Write (`*v1 = v2`) 22
Figure 5.3: Constant Assignment (`v1 = 0x********`) 23
Figure 5.4: Constant Assignment (`v1 = 0x00******`) 23
Figure 5.5: Variable Assignment (`v1 = v2`) 24
Figure 5.6: Increment (`v1++`) 24
Figure 5.7: Addition (`v1 = v2 + v3`) 25
Figure 5.8: And (`v1 = v2 & v3`) 26
Figure 5.9: Shift Left (`v1 = v2 << v3`) 28
Figure 5.10: Branch Always (`jump T1`) 29
Figure 5.11: Branch Equal (`if (v1 == v2): jump T1, else T2`) 31
Figure 5.12: Function Calls (`call FUNC`) 33
Figure 5.13: System Calls (`syscall NUM`) 35

Figure 7.1: Group 1 (Untrained) Matches by Number of Restores 54
Figure 7.2: Group 1 (Trained) Matches by Number of Restores 62
Figure 7.3: Group 2 Matches by Number of Restores 67

LIST OF TABLES

Table 7.1: Gadget Query Set 51
Table 7.2: Group 1 (Untrained) Matches 55
Table 7.3: Group 1 (Trained) Matches 63
Table 7.4: Group 1 Untrained vs. Trained Matches 65
Table 7.5: Group 2 Matches 68

ACKNOWLEDGEMENTS

This work began as a course group project, and I grateful to my lab partner, Erik Buchanan, for his extraordinary efforts in manually patching together our original buffer overflow payloads, assisting with static assembly code analysis, and writing related support tools for our research. I also thank Rick Ord for his helpful discussions and insight regarding SPARC internals for this work, and Brian Kantor and Bill Young for accommodating and supporting the varied and sometimes difficult hardware needs of the project. I am indebted to my committee members, Hovav Shacham, Stefan Savage, and Geoffrey Voelker, for their enormous amount of assistance, feedback, and encouragement throughout the development of this work.

Finally, I am forever grateful to Robin Chin for her support and patience throughout the duration of graduate school.

The text of this thesis, in full or in part, is a reprint of the following material with the full permission of all co-authors of the paper:

- Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage, "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," *In Proceedings of the 15th ACM Conference on Computer and Communications Security 2008*, pages 27-38. ACM Press, Oct. 2008.

Chapter 1, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 2, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 4, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 5, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

ABSTRACT OF THE THESIS

Finding the Bad in Good Code:
Automated Return-Oriented Programming Exploit Discovery

by

Ryan Glenn Roemer
Master of Science in Computer Science

University of California San Diego, 2009

Professor Stefan Savage, Chair
Professor Hovav Shacham, Co-Chair

This thesis investigates the pervasiveness and widespread applicability of "return-oriented programming" — an exploit technique whereby W⊕X system protections are evaded by careful stack injections (without code) that cause a vulnerable program to execute pre-existing sequences of runtime code, which in the aggregate form arbitrary computations.

In this thesis, we demonstrate that this attack is not limited to the x86 architecture, its original platform of introduction, and can be fully implemented on an architecture as completely different as SPARC. We present automated search tools that effectively and efficiently find full or partially Turing-complete return-oriented "gadget" sets in arbitrary binaries using a dedicated and extensible query language. We discuss our results from searching thousands of previously unknown binaries and find that potentially exploitable return-oriented gadgets are prevalent in the wild. Finally, we pose that the risks from return-oriented programming are fundamental

and general — our data indicates that the threat of finding a Turing-complete gadget set in an arbitrary binary is correlated simply with instruction count, and not any unique or special aspects of the target program.

Chapter 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 [20, 9, 10], 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 [11], Linux [19, 16], Mac OS X, and OpenBSD [13, 14] now support it.

However, in 2007 Shacham demonstrated that W⊕X protection could be entirely evaded through an approach called *return-oriented programming* [18]. 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 contend that this impression is incorrect on a number of levels.

First, we argue in Chapter 4 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 in Chapter 5 that the return-oriented model is not limited to the x86 ISA or even variable-length instruction sets in general. 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, 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.

Third, while previous efforts at finding gadget libraries have been laborious

and complex (usually involving static assembly code analysis), in Chapter 6 we introduce an automated return-oriented exploit discovery tool. Our tool uses customizable and extensible "gadget" queries to search a target binary for a catalog of gadgets that can be readily used in return-oriented exploits. Moreover, such searches are *fast* — typically on the order of a couple minutes.

Finally, we contend that return-oriented vulnerabilities, in the form of both partial and fully Turing-complete gadget sets, are widespread.[1] Our core return-oriented programming thesis is that given a sufficient corpus of code in a target binary, arbitrary computation is possible. Put another way, we argue that code size determines the likelihood of finding return-oriented exploits, and past certain instruction count thresholds, there are good chances of discovering a usable and possibly Turing-complete gadget set in an arbitrary binary. With this thesis in mind, we discuss our experiences performing automated gadget searches on nearly 2,000 binaries and libraries in Chapter 7. Our results provide strong support for our thesis, with substantial portions of our binary groups (at various instruction count ranges) yielding full or partially Turing-complete gadget sets.

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. Moreover, we suggest that the barrier to discovering and utilizing return-oriented exploits is significantly lowered with automated return-oriented programming tools — perhaps even to the point of equivalence with shellcode injection — the very attack vector that W⊕X set out to thwart in the first place.

In the remainder of this paper, we will provide a brief historical background of vulnerabilities and defenses leading up to W⊕X, an overview of the SPARC architecture and applying return-oriented principles to SPARC, and a discussion of our manual search for SPARC gadgets and resulting gadget catalog. We then describe our automated gadget search tool and experimental results from running the tool

[1] As described in more detail in subsequent chapters, the exploitation of a return-oriented vulnerability necessarily presupposes an initial entry vector into a vulnerable program, such as a heap or buffer overflow. The presence of return-oriented gadgets alone in a binary is not sufficient to actually compromise the runtime stack, etc. of the target.

on many binaries, and conclude with the thoughts on the future implications of the return-oriented programming model.

Acknowledgement

This chapter, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 2

Evolution: From Stack-Smashing to Return-Oriented Programming

The development of program subversion attacks and defenses have historically been a cat and mouse game. Early stack buffer overflow attacks that focused on injecting bad code to cause malicious behavior were followed by defenses that identified and mitigated the code itself. In response, attackers turned towards means other than explicit code injection to bring about the same malicious behavior in vulnerable programs. It is the latest of these techniques, return-oriented programming, which we examine as a fundamental challenge to the notion that defenses against bad *code* can ultimately protect against bad runtime *behavior*.

2.1 Traditional Stack-Smashing and Defenses

Unchecked buffer overflows have a well-documented history [7] as the most common and pervasive form of software vulnerability for non-type-safe software (*e.g.*, C/C++). Originally, buffer overflow exploits simply overwrote one or more stack frames with a new return address to divert program control to executable code placed on the stack during the same overflow ("stack-smashing") [1]. Exploit techniques quickly evolved from basic stack-smashing to exploiting the heap [6], `malloc()`/`free()` errors [2], integer overflows [3], and pointers [17].

Defenses to these attacks are broadly classified as (1) stack-smashing prevention, or (2) non-executable stack protection. The first approach attempts to prevent the initial buffer overflow with, *e.g.*, pointer integrity checks or stack "canaries" to detect if the stack has been smashed [7].

The second approach accepts the possibility of a buffer overflow, but limits the ability of the attacker to execute malicious code. W⊕X is a version of this defense that ensures that process memory is either writable or executable, but not both (hence "W⊕X"). Systems implementing W⊕X (in some manner) include Solar Designer's StackPatch [19], PaX on Linux [16], W^X on OpenBSD (introduced in 3.3 [13], and expanded to x86 in 3.4 [14]), and non-executable stacks on Solaris/SPARC [12] (introduced in Solaris 2.6). This thesis focuses only this second defense (prevention of bad code injection), specifically examining the non-executable stack protection on Solaris/SPARC.

2.2 W⊕X and Return-to-Libc

While W⊕X prevents execution of code injected on the stack, buffer overflow and related attacks can nonetheless compromise program control through other means. Attacks can still cause stack frames to modify registers (including the program counter, the stack pointer, and the frame pointer) in a way as to effectively control a process. For example, if an attacker can overflow a stack frame, they can set the return address of the frame to point to other executable code (outside of the process), such as a loaded library function. When the stack frame returns, the code set by the attacker is run, and the program is effectively hijacked without the attacker injecting any executable code on the stack.

The classic form of this attack overflows a stack buffer to cause a return from the stack frame to point the program counter to a function in libc, the standard C library, and is commonly known as a "return-to-libc" attack. Libc is a popular target for attacks because: (1) it provides a plethora of useful functions, including many wrapped system calls (*e.g.*, `system()`, `exec()`, `fork()`), and (2) virtually every program on a Unix-like system will load libc. Solar Designer published the

first widely acknowledged return-to-libc exploit on Linux/x86 in 1997 [20], which has subsequently spread widely on the x86 and other platforms.

2.3 Return-Oriented Programming on x86

Most return-to-libc attacks overflow a few stack frames and divert program control to a libc function like `system()` or `exec()`. While simple and effective, this approach depends on the presence of the called function(s) in the target library, and that the program registers are set to a usable state for the function call. Extending the attack beyond a single function jump, exploit designers have used additional stack frames to jump into short sequences of libc code to set up registers for a subsequent return into a full libc function (on x86-64 [9] and SPARC [10]).

In 2007, Shacham extended this concept into a class of return-to-libc attacks that relied entirely on short instruction sequences without any direct function calls [18]. Shacham's technique demonstrated that short sequences of code in a library like libc could be aggregated to provide the full power of arbitrary computation during an exploit without ending in the usual system or function call. By combining instruction sequences into intermediate units of computation, an attacker could construct a payload of several exploit stack frames (each pointing to an instruction sequence) that would effectively become an arbitrary "program".

Shacham combed the entire instruction byte stream of libc on x86/Linux and identified "useful" instruction sequences followed by a return instruction. The short instruction sequences were grouped together to form a catalog of "gadgets," each performing a useful unit of computation. The catalog includes gadgets for memory operations (load/store), basic arithmetic and logic operations (add, subtract, not, and, shift, etc.), control flow (unconditional and conditional jumps), and system and function calls. An attacker with access to the gadget catalog could construct a specific buffer overflow payload comprised of gadgets that cause the program counter to return to a specific libc instruction sequence for each exploited stack frame, after the initial stack overflow. Shacham noted that his collection of x86 gadgets was Turing-complete, and provided, in essence, a stripped-down programming paradigm,

facetiously dubbed "return-oriented programming".

Shacham further posed that unique aspects of the relatively unstructured x86 architecture specifically aided the return-oriented attack. Shacham conjectured that modifying the x86 architecture to be more RISC-like could provide potential avenues for defense. We explore this claim in the following two chapters, looking at buffer overflows on the SPARC instruction set architecture in Chapter 3 and find that the return-oriented programming attacks on SPARC are indeed feasible in Chapter 4.

Acknowledgement

This chapter, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 3

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 chapter only surveys the SPARC features relevant to stack overflows and program control hijacking, more detailed descriptions of the SPARC architecture are variously available [21, 22, 15].

3.1 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.

3.2 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 the purposes of this thesis, 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.

3.3 The Stack and Subroutine Calls

The basic layout of the SPARC stack is illustrated in Figure 3.1. 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 thesis.

3.4 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 proof-of-concept return-to-libc exploit on Solaris 2.6 / SPARC [10], modeled after Solar Designer's original exploit. McDonald

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 3.1: SPARC Stack Layout

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.

Acknowledgement

This chapter, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 4

Return-Oriented Programming on SPARC

Like other modern operating systems, Solaris includes an implementation of W⊕X [12], supported by page-table hardware in the SPARC processor. In this chapter 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 discussed in Section 3, 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.

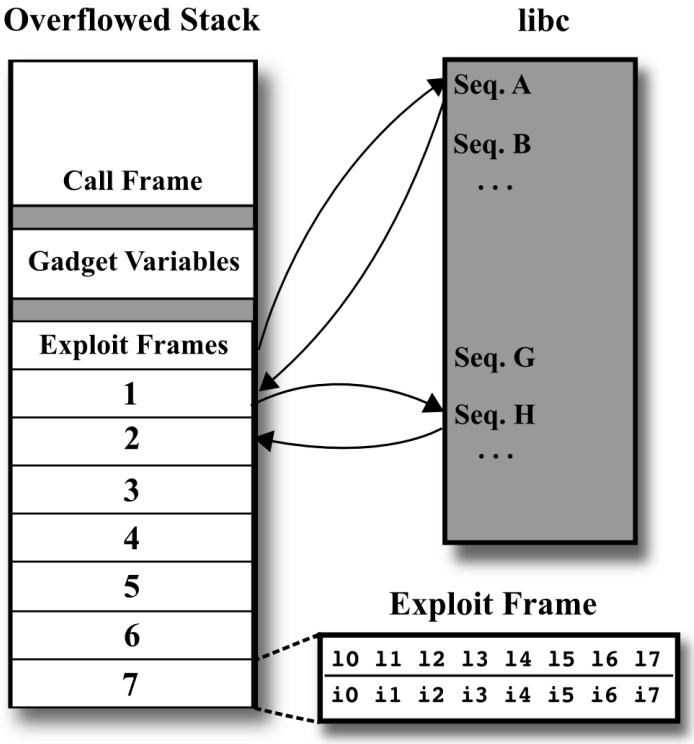

Figure 4.1: Return-Oriented Program

A return-oriented program is really a carefully packed exploit string buffer. Once delivered via a stack overflow, the program operates as illustrated in Figure 4.1. Packed exploit frames contain register values that influence program control to jump into short instruction sequences in a target binary, in the most common case, a loaded library like libc. Once a given instruction sequence finishes and returns, the next exploit frame loads new register values and jumps to a different instruction sequence in the target binary. By piecing together instruction sequences, we form gadgets that 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. (Figure 4.1 also depicts gadget variable storage and the function call gadget stack frame, which will be explained later).

To provide sufficient background in return-oriented programming on SPARC, we initially examine the Solaris C Standard Library to discover and describe a Turing-complete set of SPARC gadgets using only manual code analysis. In later chapters of this thesis, we will describe automated search tools that can quickly discover usable gadgets with heuristic queries and comment on the ubiquity of return-oriented attacks in various target binaries.

4.1 Finding SPARC Instruction Sequences in libc

We 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.

4.2 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 a target binary (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).

4.3 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, target 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.

4.4 Gadget Abstractions and Practical Return-Oriented Programming

In Chapter 5, we enumerate a Turing-complete set of gadgets that perform a useful collection of basic computational operations. While the address entry points and restored register values can be coded by hand into a string buffer exploit payload (and indeed we did so in our original research), the complexity of a return-oriented program is limited by the practical difficulties of such an exercise.

In 2008, Buchanan, et al. [4] introduced an approach to abstracting return-oriented program creation. The authors wrote an application programming interface (API) that provides C functions for setting gadget variable values and calling gadget operations. The C API then provides means to easily pack up a string buffer exploit payload that can hijack a vulnerable program to run the return-oriented program attack.

The authors further abstracted the exploit process by creating a compiler that views the C API as an instruction set architecture in its own right, where each API gadget operation is analogous to an assembly instruction. The compiler implements a subset of the C programming language, and translates each line of code into gadget variable and operation C function calls using the C API.

Using these multiple abstraction layers, complicated return-oriented programs such as a basic selection sort can be easily constructed in a high-level C-like language,

then passed through the compiler to the API to an ultimate complex string buffer exploit payload that can effectively invoke arbitrary behavior in a target vulnerable program. Thus, although this thesis focuses on a low-level portion of the world of return-oriented programming — the gadget catalog search — the security threat from a discovered set of gadgets for a given target binary is real, as once API and compiler abstraction layers are hooked into a new set of underlying gadgets, complex and powerful return-oriented attacks can be quickly and easily created.

Acknowledgement

This chapter, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

Chapter 5

SPARC Gadget Catalog

In this chapter, we describe an exemplary set of SPARC gadgets found by manual investigation of the Solaris standard C library. Chapter 6 expands this discussion to automatically discovering gadget catalogs with our new general-purpose search tool.

Our collection loosely mirrors Shacham's x86 gadget catalog [18], 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 in this chapter has been successfully implemented in a higher-level gadget C API and exploit compiler and tested in real return-oriented exploits [4].

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 target binary (libc) instruction sequence assembly code.

5.1 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.

5.1.1 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 5.2.1.

5.1.2 Pointer Read

The pointer read gadget (`v1 = *v2`) uses two instruction sequences and is described in Figure 5.1. 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 5.1: Pointer Read (`v1 = *v2`)

5.1.3 Pointer Write

The pointer write gadget (`*v1 = v2`) uses two sequences and is described in Figure 5.2. 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 5.2: Pointer Write (`*v1 = v2`)

As the second instruction sequence indicates, we were not always able to find completely ideal assembly instructions in our target binary (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.

5.2 Assignment

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

5.2.1 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 5.3. Non-zero hexadecimal byte values are denoted with "**".

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

Figure 5.3: 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, Figure 5.4 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 5.4: Constant Assignment (v1 = 0x00******)

5.2.2 Variable Assignment

Assignment from one gadget variable to another (v1 = v2) is described in Figure 5.5. 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 5.5: Variable Assignment (v1 = v2)

5.3 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.

5.3.1 Increment, Decrement

The increment gadget (v1++) uses a single instruction sequence for a straightforward load-increment-store, as shown in Figure 5.6. The decrement gadget (v1--) consists of a single analogous load-decrement-store instruction sequence.

Inst. Seq.	Preset	Assembly

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

Figure 5.6: Increment (v1++)

5.3.2 Addition, Subtraction, Negation

The addition gadget (`v1 = v2 + v3`) is shown in Figure 5.7. 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`.

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 5.7: 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.

5.4 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.

5.4.1 And, Or, Not

The bitwise and gadget (`v1 = v2 & v3`) is described in Figure 5.8. 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 ...`

Figure 5.8: 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`.

5.4.2 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 5.9. 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.

5.5 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 (mir-

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 5.9: Shift Left (`v1 = v2 << v3`)

roring 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.

5.5.1 Branch Always

The branch always gadget (`jump T1`) uses one instruction sequence consisting of a `ret`, `restore`, as shown in Figure 5.10. 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 5.10: Branch Always (`jump T1`)

5.5.2 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 5.11. 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`).

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

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

5.6 Function Calls

Virtually all public return-to-libc SPARC exploits already target libc function calls. As our target library for this chapter is libc, we easily provide similar abilities with our function call gadget. For gadget collections targeting binaries other than libc, the function call gadget may call any function that is available at runtime to the target binary.

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 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 4.1). 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 Figure 5.12, 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 5.12: Function Calls (call FUNC)

5.7 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 5.13. 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.

Acknowledgement

This chapter, in part, is a reprint of the material as it appears in the Proceedings of the 15th ACM Conference on Computer and Communications Security 2008, by Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. The thesis author was the primary investigator and author of the reprinted portions of this paper.

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 5.13: System Calls (`syscall NUM`)

Chapter 6

Automated Gadget Searching

The 19 gadgets comprising the gadget catalog described in Chapter 5 were discovered over the course of nearly three weeks of manual assembly code analysis of the specific Solaris Standard C library binary we targeted. However, such a discovery approach is clearly not scalable — for libc alone on the Solaris/SPARC platform, there are many variants of versions and producers, adding to the fact that not all Solaris binaries necessarily load libc.

Thus, the gadget catalog in Chapter 5 is really a starting point for two follow-on questions: (1) How widespread is the return-oriented attack across other versions of libc and entirely different Solaris binaries and libraries? (2) How can one quickly (but roughly) determine if a given binary is vulnerable to a return-oriented attack? With these questions and issues in mind, we present a gadget search tool that automates the gadget discovery process to enable rapid heuristic analysis of return-oriented vulnerabilities in arbitrary Solaris binaries and libraries.

This chapter describes the use, design, and limits of the search tool. Chapter 7 discusses our vulnerability measurements over several large sets of Solaris binaries and libraries using the search program and supporting reporting tools.

6.1 Gadget Search Tool

We introduce "`traitor`," a search tool for discovering return-oriented "gadgets" and "instruction sequences" in an arbitrary target binary. `traitor` is written entirely in Python and presently only implements assembly code analysis for the SPARC instruction set. Modules for other instruction set architectures (like x86 or PowerPC) could be developed and added to the tool.

The `traitor` program takes as input a target binary and a gadget query file. The gadget query file specifies the assembly instruction sequences that comprise return-oriented gadgets in a permissive manner (allowing regular expressions and alternative "fall-back" queries). `traitor` then loads in the assembly of the target binary and heuristically matches gadget queries over the entire target instruction corpus, and reports all matched gadgets.

Search query specifications and the matching algorithm are described in more detail in the subsequent sections of this chapter. However, it is important to note that the accuracy of gadgets discovered by `traitor` depends on the quality and correctness of both the query and the search algorithm. While the search algorithm correctness is an implementation detail of `traitor`, the quality of the query specification is ultimately the responsibility of the search tool user. From our experience testing and performing vulnerability measurements with `traitor`, we pose that our queries and the underlying search algorithm are positively correct to the extent that if `traitor` reports a gadget matching a given search query, then the reported instruction sequences could be used in a subsequent return-oriented attack or wrapped into a practical return-oriented API / compiler exploit tool. However, even with proper queries, `traitor` is vulnerable to false negatives (missing potentially usable instruction sequences for a gadget). Because `traitor` is a heuristic tool, we made the design decision to take a conservative approach to some aspects of gadget matching (described later), at the expense of excluding otherwise viable gadgets.

6.2 Search Queries

A search query specifies a general notion of what the assembly instruction sequences that comprise a given gadget should look like. The query specification for `traitor` is simply a Python file containing objects and collections that correspond to Python query classes.

A search query consists of a "library" which contains multiple "gadget lists". A gadget list is a collection of alternative queries for essentially the same gadget, so that if `traitor` fails to find one version, it can revert to backup gadget queries. Naturally, a gadget list contains multiple "gadget" queries. The gadgets are composed of "sequence" queries, which are further composed of "instruction" objects. Instruction queries specify an assembly opcode search term and contain multiple "argument" objects — the most basic query unit.

The query language aims to allow a user to specify any arbitrary gadget with enough precision to guarantee that a match in a target library for a properly specified query should be usable in a return-oriented exploit. In this fashion, the query language permits extensible vulnerability analysis for unknown gadgets as well as new target binaries or libraries.

The `traitor` tool suite includes a basic query specification file, "`query.py`," containing search queries for 19 core gadgets from our original gadget catalog in Chapter 5. `traitor` was in fact developed against the target libc binary used in our original manual gadget search, although "`query.py`" contains broader backup queries than those that are known to match the target libc. For a rough performance comparison, our manual target libc search for gadgets took around three weeks of effort, while `traitor` replicates the search in less than 500 seconds running on a server-grade computer.

6.2.1 Arguments

Basic Arguments

The starting point for a gadget search query is the argument. An argument is simply an object that takes a regular expression string for a match. For example:

```
A("%i1")
A("%(i[0-5]|l[0-7])")
A(".*")
```

would all match the register `%i1`. (Note that the `Argument` class is aliased to `A`).

In this fashion, a search query can specify any assembly instruction argument, from registers to immediate values. Argument declarations can be used in an instruction query to match an entire instruction. For example:

```
I("add", [A(".*"), A(".*"), A(".*")])
```

matches any of the following instructions:

```
add %i1, %i2, %i3
add %i0, %i2, %i0
add %l1, 0x12, %l2
```

(Note that the `Instruction` class is aliased to `I`).

When declared as a variable, an argument object has the additional property that the same argument object instance must match the *same* piece of assembly code at both the instruction and sequence level. Thus, if we restructure the example instruction query as:

```
ARG1 = A(".*")
I("add", [ARG1, A(".*"), ARG1])
```

then the first and third arguments must be the same. This means of the three previous example matched instructions, now only

```
add %i0, %i2, %i0
```

matches (because `%i0` is the same for the first and third arguments).

Arguments "Plus"

We also provide an augmented Argument class, `ArgumentPlus` (aliased to `AP`). This special argument class takes a register pattern, such as "`%i[0-7]`" and appends an extra pattern to allow an extra hexadecimal number to appear in the matched results. For example, the query:

```
ARG1 = AP("\%i[0-7]")
```

would match all of the following assembly code snippets:

```
%i0
%i1 + 0x1
%i1 + 0xdecafbad
```

which is particularly useful for memory instructions like loads and stores that often access memory at a register-stored address plus a numerical offset.

The special aspect of this argument class is that for "same argument matching," only the register portion is compared for the match. Thus, revising a previous query:

```
ARG1 = AP("\%i[0-7]")
I("ld", [M(ARG1), ARG1])
```

would successfully match all of the following:

```
ld [%i0], %i0
ld [%i1 + 0x123a], %i1
```

by allowing the extra numerical part on matching, but ignoring it for the same argument requirement. Thus, the augmented argument class permits flexibility for matching, but provides the necessary structure to ensure control flow through designated registers.

Memory Arguments

The basic argument object can handle most argument forms (*e.g.*, registers, immediate values, labels) without incident. However, memory-based arguments have a special wrapper class, `MemoryWrapper` (aliased to `M`), which wraps an argument object to match arguments comporting with the SPARC assembly memory format. For example, using variable-matching and a memory wrapper, we can specify a query:

```
ARG1 = A(".*")
I("ld", [M(ARG1), ARG1])
```

which would match instructions like:

```
ld [%i0], %i0
```

but not:

```
ld [%i0], %i1
ld [%i0], %o0
```

because the first and second registers are different.

Cross-Sequence Arguments

Variable matching allows argument object queries to all get the same match *within* a sequence query, but a query may need to specify an argument query *across* different sequences. However, the register window slides on every return, such that an input register in one exploit frame / sequence becomes an output register in the next.

To address this challenge, the input / output wrapper class, `InOutWrapper` (aliased to `IO`), wraps an argument object to permit a regular expression that matches an input (`%i`) register in one sequence to match the same regular expression in the next, but adjusted to the output (`%o`) register.

Thus, we can create an example query (aliasing `Gadget` to `G` and `Sequence` to `S`) as follows:

```
ARG1 = A(".*")
G([
    S([ I("ld", [M(A("%i.*")), IO(ARG1)]),
        I("ret"),
        I("restore"),
    ]),
    S([ I("ld", [IO(M(ARG1), "o"), A("%o.*")]),
        I("ret"),
        I("restore"),
    ]),
])
```

which would match a first sequence (with `%i0` matched for the input / output wrapper on `ARG1`):

```
ld       [%i0], %i0
ret
restore
```

and a second sequence (now matching `%o0` for the input / output wrapper on `ARG1`):

```
ld       [%o0], %o1
ret
restore
```

As the above example shows, the input / output wrapper can be used to not only wrap basic argument query objects, but memory wrapper objects as well, enabling more specific and powerful querying.

6.2.2 Instructions

The next abstraction level is the instruction query object, which is intended to match assembly instruction lines in the target binary. An instruction query consists of an opcode (regular expression) and one or more argument query objects. Thus, a basic instruction query like:

```
I("add(|cc)", [A("%i.*"), A("0x[0-9]"), A("%i.*")])
```

would match any of the following:

```
add      %i1, 0x1, %i1
addcc    %i1, 0x4, %i2
```

Intermediate Instructions and Depth Limits

Instruction queries specify a match of one line of assembly code in the target binary. However, the query object has additional parameters that can affect the search algorithm with respect to additional instruction queries.

One issue with conducting matches on instruction sequence queries is whether to allow intermediate, non-matching instructions between matches. For instance, if the query sequence:

```
S([ I("ld", [M(A("%o.*")), A("%i.*")]),
    I("ret"),
    I("restore"),
])
```

encounters the following assembly code:

```
ld       [%o0], %i0
sub      %i1, %i2, %i3
ret
restore
```

should it allow the intermediate "`sub`" instruction and consider the assembly sequence a match?

The search algorithm (discussed in detail in Section 6.3) permits intermediate instructions that pass a number of criteria. On the query side, one parameter that can be specified at the instruction query level is how many intermediate instructions to allow until the next matched instruction. This parameter, called the "depth," defaults to zero, disallowing any intermediate instructions. So our example query above would *not* match the assembly code. However, setting the depth parameter to one or more would allow the "`sub`" instruction as a non-matching intermediate, *e.g.*:

```
S([ I("ld", [M(A("%o.*")), A("%i.*")], depth=2),
    I("ret"),
    I("restore"),
])
```

would produce a sequence match. Thus, by using the depth parameter a `traitor` user can control the potential runtime of the search (by limiting depth between instructions), and enforce instruction proximity (*e.g.*, requiring that a matched `ret` is immediately followed by a `restore`).

Destination Register Protection

The next complication with intermediate (non-matching) instructions is when an otherwise permissible intermediate instruction overwrites a destination register used by an earlier matching instruction in a sequence. For example, consider the situation where our query:

```
ARG1 = A("%i.*")
S([ I("ld", [M(A("%o.*")), ARG1]),
    I("st", [ARG1, M(A("%o.*"))]),
    I("ret"),
```

```
    I("restore"),
])
```

encounters the following assembly code:

```
ld       [%o0], %i0
sub      %i1, %i2, %i0
st       %i0, [%o1]
ret
restore
```

The variable `ARG1` successfully matches register `%i0` for both the `ld` and `st` instructions. However, the value loaded into `%i0` by the `ld` is not correctly stored in the `st` instruction because the intermediate `sub` instruction overwrites the value in `%i0`.

To address this situation, instruction queries take an additional parameter to designate how many subsequent matching instructions to protect a destination register (if there is one). By default instruction queries protect destination registers for every subsequent instruction in a sequence query. Thus, our above example would not match, because by default the destination register for the `ld` instruction is protected all the way through the last instruction query in the sequence. If destination register protection is set to -1, then there is no protection against any subsequent matching or non-matching instructions. If protection is set to zero, then the destination register is protected up to, but *not* including, the next matching instruction. With this query feature, the query writer can specify at a very granular level how far to protect destination registers so as to allow a good degree of balancing the goals of maximizing possible matches and ensuring that assembly code matches are indeed usable sequences.

Branch Instructions

The basic instruction query can handle all instruction searches except for branches. As discussed in Section 5.5, the branch instructions likely sought for a return-oriented attack are annulled branches that branch forward two instructions. Accordingly, the branch instruction query, takes an opcode query, an opcode annulled query (typically "*a*"), and a branch displacement integer (with a default value of two

instructions). The resulting query gives the query writer control over the attributes of branch instructions used in a return-oriented control flow operation with a high degree of precision.

6.2.3 Sequences, Gadgets, and Gadget Lists

Looking above the instruction query level, a sequence query is a list of one or more instruction queries. A sequence query can take an additional parameter that limits the number of returned results for queries that have the potential to return an excessive number of results.

Similarly, a gadget query is a list of one or more sequence queries. A gadget query can also specify a parameter to globally limit the maximum number of assembly code sequences a sequence query can match.

For a given gadget, it is often the case that one form of a gadget query will find no matches in a target binary, while an alternate (but functionally equivalent) gadget query would match. To address this situation and provide fallback flexibility for `traitor`, the gadget list abstraction is provided. A gadget list query comprises one or more gadget queries, such that matching any one of the individual queries is considered an overall "match" for a given gadget.

For example, the variable assignment gadget (*see* Section 5.2.2) requires one `ld` followed by a `st` instruction, but these could occur over one or two instruction sequences. Thus a gadget list of two gadget queries would be appropriate for this situation as follows (note `GadgetList` is aliased to `GL`):

```
REGS = "%(i[0-5]|l[0-7])"
ARG1 = A(REGS)
GL([
    G([
        S([
            I("ld(|uw)", [M(A(REGS)), ARG1]),
            I("st(|w)",  [ARG1, M(A(REGS))]),
            I("ret"),
            I("restore"),
        ]),
    ]),
```

```
    G([
        S([
            I("ld(|uw)", [M(A(REGS)), IO(ARG1)]),
            I("ret"),
            I("restore"),
        ]),
        S([
            I("st(|w)",  [IO(ARG1, "o"), M(A(REGS))]),
            I("ret"),
            I("restore"),
        ]),
    ]),
])
```

6.3 Search Algorithm

At a high level, `traitor` takes two inputs: a disassembled target binary and a search query. The disassembled target binary is internally re-disassembled to gather additional information from the machine code binary representation. The assembly instruction corpus and search query objects are then passed to a matching engine which searches the instruction list for query matches starting at the argument level and building all the way up to the full gadget queries.

As a single instruction sequence query can yield assembly code matches that overlap, it is easy to see that the runtime of the actual search has the potential to quickly become intractable. Consequently, the search algorithm is built around a number of design decisions and heuristics that attempt to ensure that the search runs are computationally feasible and produce valid assembly code results. Additionally, as Section 6.2 describes, the query classes provide parameters at various abstraction levels that facilitate filtering and limiting the actual searches performed by `traitor`.

6.3.1 Gadget Matching

The basic target of a search is a gadget, which is a series of instruction sequences that matches a gadget query. Multiple gadget queries may be wrapped in a gadget list for a given search, in which case the reporting tools for `traitor` report

a gadget "match" if any of the gadget queries result in at least one match. In either case, the basic algorithm for a single gadget query search is as follows:

Each instruction sequence query in the gadget query is matched to assembly code instruction sequences in the target binary. Section 6.3.2 describes this process in detail.

Each set of assembly code sequences matching a sequence query is then limited down to a maximum number specified at either the sequence or gadget query level. The underlying issue is that for multi-sequence gadget queries, if a large number of assembly code sequences match the individual sequence queries, the overall complexity can quickly become intractable when testing all possible sequence combinations.

However, a converse problem is that too few matched assembly code sequences can possibly make a gadget search fail when there were in fact underlying sequences to match the full gadget. Accordingly, a heuristic approach is used to intelligently choose the "best" assembly code sequence candidates up to the limit value. The search algorithm thus prefers argument queries that are input / output wrapper objects and then tries to find the widest dispersion of different register values. In this manner, the heuristic attempts to give as many options (within the sequence search result limits) for cross-sequence compatibility.

All permutations of the assembly code sequence matches for each sequence query comprising the overall gadget query are then inspected. Sequence combinations are mainly checked to verify that any input / output argument wrapper constraints are met. If the query constraints are met, then the given assembly code sequence combination is a valid match, and should be usable in a return-oriented exploit for the given gadget.

6.3.2 Instruction Sequence Matching

The heart of the search algorithm for `traitor` lies in the instruction sequence search. The matching engine takes each sequence query, searches through the list of assembly code instructions for the target binary for matches, and produces a list of zero or more assembly code instruction sequences matches. The sequence matching

algorithm is as follows:

The list of assembly code instructions is searched linearly front-to-back, looking for matches for the last instruction query in the sequence query. When the last instruction query is matched to an assembly code instruction, a recursive search begins, moving backwards through the list of instruction queries comprising the sequence query. Because there may be multiple overlapping assembly code sequence matches that terminate at the same last assembly code instruction, a list of all possible assembly code sequences is returned. At this point, the only requirement is that all individual instruction queries are matched for a given sequence. The limiting factor on how far back from the last matching instruction to recursively search is the sum of all instruction query depth parameters. In this manner, the search space for a given entry point last instruction is dramatically reduced. In the common case — four instruction queries, two with a depth of five, two with a depth of zero — the search space from each last matching instruction is 14 instructions. For a reasonable target code corpus containing 4,000 `restore` suffixes, this yields a total search space of about 50,000 instructions for an average sequence query.

Each set of possible matching sequences is then checked for compliance with the query parameters and return-oriented program constraints. Each sequence is first checked against all instruction query depth limitations. Next, all intermediate (non-query-matching) instructions are checked against a list of disallowed instructions. Instructions in this list include memory operations, branches, traps, jumps, function calls, etc. The disallowed list is over-inclusive, but ensures that no unintended memory access, traps or control flow operations occur.

Argument queries are then checked for matched instruction arguments to validate that same-variable requirements (*e.g.*, two arguments must match the same register) are met. For each instruction query that has enabled destination register protection, the matched destination registers are examined to ensure that no later intermediate instruction overwrites a protected destination register.

At this point, each remaining instruction sequence should be valid — capable of being used in a return-oriented exploit, provided the original query was correctly constructed. As an optimization step, identical sequences — sequences result-

ing in identical machine code for each instruction — are removed, so that only unique matches remain. The main reason to return multiple matching assembly code sequence matches in the first place is to permit greater leeway in matching constraints across sequences, so one identical sequence adds no additional benefit over another. As an additional optimization, sequence match results for a given query are cached, using the query as a lookup key. Because many sequence queries are reused across several gadgets, this provides a speedup for the core `traitor` search task — conducting the recursive instruction sequence searches.

The list of assembly code sequence matches that pass all of the above tests and filters are returned for gadget matching, as described in Section 6.3.1.

Chapter 7

Vulnerability Analysis

The `traitor` search tool and default gadget query was designed against the Solaris 10 libc we originally targeted in our manual analysis from Chapter 5. However, the value and wider impact of `traitor` depends on its ability to find and identify gadgets in arbitrary target libraries and binaries, and not those that are already known to have a Turing-complete gadget set. Thus, this chapter utilizes `traitor` on a large number of Solaris SPARC binaries and describes the gadget search results, as well as metrics and implications for return-oriented programming as a general threat.

7.1 Measurement Methodology

Our basic approach is to run `traitor` on a large number of target binaries, searching for a core list of 19 gadgets (as gadget list queries). The target gadget query set is taken from our gadget catalog described in Chapter 5 and is enumerated in Table 7.1.

Our default query file ("`config.py`") contains several gadget list queries for each of the 19 gadgets we seek, ranging from condensed "optimal" gadget queries to much more permissive, but "clunky" queries. In most cases, this means that the tightest gadget queries involve many fewer instruction sequences than the permissive ones. Most gadget list queries have two to four individual gadget queries, with our

Table 7.1: Gadget Query Set

	Gadget	**Example**
0	Nop	
1	Memory Load	`v1 = *v2`
2	Memory Store	`*v1 = v2`
3	Assignment	`v1 = v2`
4	Constant Assignment	`v1 = 0x00******`
5	Increment	`v1++`
6	Decrement	`v1--`
7	Negation	`v1 = -v2`
8	Addition	`v1 = v2 + v3`
9	Subtraction	`v1 = v2 - v3`
10	And	`v1 = v2 & v3`
11	Or	`v1 = v2 | v3`
12	Not	`v1 =  v2`
13	Branch Always	`jump T1`
14	Branch Equal	`if (v1 == v2): jump T1, else T2`
15	Branch LTE	`if (v1 <= v2): jump T1, else T2`
16	Branch LT	`if (v1 < v2): jump T1, else T2`
17	Function Call	`call FUNC`
18	System Call	`trap NUM`

most difficult gadget searches rounding things out with five queries for the bitwise not gadget and seven queries for the system call gadget. We relaxed the requirements for some of our system call gadget queries by allowing the sequence to not properly return for the next gadget. This means that system call gadget matches for these queries can only be used to trap to a call like `exec`, which does not return, or the exploit writer is willing to allow the return-oriented program to crash after the system call. All in all, the default query file provides a reasonable amount of flexibility for finding the 19 core gadgets. For the original target libc, we match at least one gadget query for each of the 19 base gadget list queries, and within each gadget list, most individual gadget queries are matched as well. We describe our results with this early query file on other binaries in the subsequent sections of this chapter.

In addition to the basic search program, our search tool suite provides supporting scripts and programs that facilitate downloading and disassembling large numbers of target binaries. Our reporting and aggregation tools run `traitor` on the disassembled target binaries, produce individual target search reports, and aggregate all search reports for a given binary group into a single, comprehensive report.

Our experiment consists of three stages. First, we download and search a number of target binaries using the default search query for our original target libc. Second, we "re-train" the gadget search queries on this group of binaries and compare resulting matches. Finally, we assess `traitor`'s performance on a new group of previously unknown target binaries using our improved and trained set of search queries.

7.2 First Binary Group, Untrained

Our target binaries are downloaded from the Sun Freeware website [5]. Our first binary group consists of 224 packages. We extract and disassemble binaries and libraries (basically everything in "`bin/`" and "`lib/`" that is a valid SPARC Solaris compiled binary) from the packages, yielding 969 individual target binaries.

Because `restore` instructions are the basis of each gadget, the total number of instructions and `restore`s are used as our base metric for evaluating matches for

a given target binary. Roughly speaking, if our conjectures that (1) return-oriented exploits are generally possible on binaries of sufficient size, and (2) `traitor` can find a reasonable subset of all possible exploits are valid, then as the number of `restore` and total instructions in a target binary increases, the likelihood that `traitor` finds more gadgets from our core list of gadgets should increase as well.

7.2.1 Results

Our search of the 969 binaries took around six hours to complete on a quad processor 3.0 GHz machine with 16 GB of memory. For most of the core gadget set, we had multiple individual gadget queries for each aggregating gadget list query. All together, our search run used 54 specific gadget queries for the core 19 gadgets in our search set. For individual binary searches, the shortest search took 0.1 seconds for a target comprising 9 `restore`s / 122 instructions, and the longest took 16 minutes for a rather large target with 30,335 `restore`s / 1,316,921 instructions. As our first binary group has a disproportionate number of small instruction-count binaries, an instructive metric is to look at the search time per instruction. The minimum per-instruction search time was 0.0003 seconds, the maximum was 0.0010 seconds, and the average was 0.0005 seconds. The overall number of instructions in a given target binary did not correlate with higher or lower per-instruction times within these ranges, suggesting that, at least for this target group, `traitor` search times scale linearly in total instruction count.[1] Thus, the average search times and our minimum to maximum times indicate that `traitor` can be efficiently run against arbitrarily-sized binaries.

Our results for the initial pass on the first binary group are illustrated in Figure 7.1. The x-axis identifies the number of `restore`s in the given target binary, and the y-axis shows the number of matches against the 19 possible gadget queries. The query results are sorted along the x-axis by number of restores from smallest (9) to largest (30,335). The x-axis distribution shows that most of the target binaries in our first group are very small, with less than 100 `restore` instructions and 20,000

[1] This is true provided that `traitor` has sufficient runtime memory. Search times exponentially increase if the target instruction set is so large as to cause the `traitor` process to swap to disk.

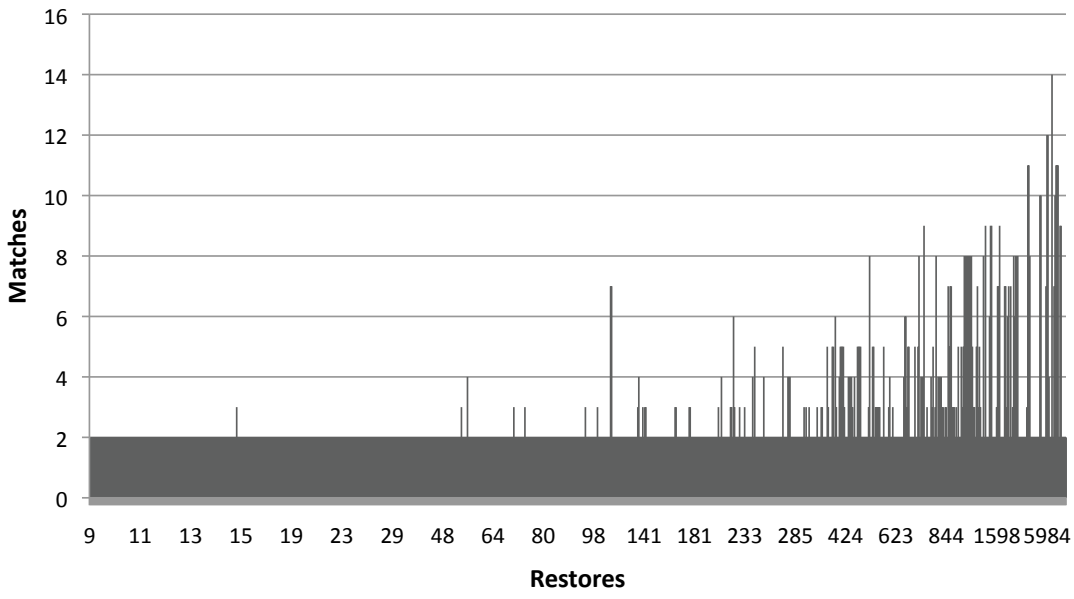

Figure 7.1: Group 1 (Untrained) Matches by Number of Restores

total instructions. (By comparison, the libc targeted in our manual search had over 4,000 `restore` instructions and 150,000 total instructions).

Looking at Figure 7.1 we note that every query we ran has at least *two* matches. These correspond to our two "trivial" gadgets, the `nop` and branch always, which are identical and comprised of `ret`, `restore` sequence. Every binary we tested has such a sequence.

Past the trivial matches, Figure 7.1 illustrates increased numbers of matches as the number of `restore`s rises. The maximum number of matched gadgets was 14 out of 19, with an average of 2.56 gadgets matched per query (although this number is depressed because of the substantial number of small binaries in our binary group). Thus, although our untrained pass was unable to match all 19 core gadgets, the results do generally indicate that `traitor` can match gadgets in arbitrary binaries and will find more distinct gadgets as the size of the target binary code corpus increases.

Table 7.2 explores the search results in more detail. The table provides the number of target binary matches for each of the 19 gadget searches. The first data

Table 7.2: Group 1 (Untrained) Matches

Restores Range		**All** (9-30335)		9-347		351-597		614-892		922-2705		2855-7971		10164-30335	
Num. Binaries		**All** (969)		729		69		59		68		30		14	
Complete		0		0		0		0		0		0		0	
Min. Matches		2		2		2		2		2		2		2	
Max. Matches		14		7		8		9		9		14		11	
Avg. Matches		2.56		2.09		3.13		3.49		4.66		4.67		5.86	
0	Nop	969	*100%*	729	*100%*	69	*100%*	59	*100%*	68	*100%*	30	*100%*	14	*100%*
1	Memory Load	38	*4%*	1	*0%*	0		6	*10%*	17	*25%*	9	*30%*	5	*36%*
2	Memory Store	1	*0%*	0		0		0		1	*1%*	0		0	
3	Assignment	113	*12%*	18	*2%*	23	*33%*	21	*36%*	34	*50%*	10	*33%*	7	*50%*
4	Const. Assn.	107	*11%*	21	*3%*	21	*30%*	22	*37%*	29	*43%*	9	*30%*	5	*36%*
5	Increment	27	*3%*	1	*0%*	7	*10%*	3	*5%*	6	*9%*	5	*17%*	5	*36%*
6	Decrement	6	*1%*	0		0		3	*5%*	0		3	*10%*	0	
7	Negation	3	*0%*	0		0		0		0		3	*10%*	0	
8	Addition	43	*4%*	2	*0%*	1	*1%*	6	*10%*	18	*26%*	9	*30%*	7	*50%*
9	Subtraction	31	*3%*	2	*0%*	1	*1%*	4	*7%*	11	*16%*	7	*23%*	6	*43%*
10	And	20	*2%*	2	*0%*	1	*1%*	0		15	*22%*	2	*7%*	0	
11	Or	17	*2%*	2	*0%*	0		1	*2%*	5	*7%*	5	*17%*	4	*29%*
12	Not	0		0		0		0		0		0		0	
13	Branch Always	969	*100%*	729	*100%*	69	*100%*	59	*100%*	68	*100%*	30	*100%*	14	*100%*
14	Branch Equal	25	*3%*	0		1	*1%*	2	*3%*	10	*15%*	6	*20%*	6	*43%*
15	Branch LTE	2	*0%*	0		0		0		0		1	*3%*	1	*7%*
16	Branch LT	4	*0%*	0		0		0		2	*3%*	1	*3%*	1	*7%*
17	Function Call	106	*11%*	13	*2%*	23	*33%*	20	*34%*	33	*49%*	10	*33%*	7	*50%*
18	System Call	0		0		0		0		0		0		0	

column shows results for all 969 binaries in the first binary group, providing absolute number of matches and percentages. The average number of matched gadgets across the entire group was 2.56 gadgets out of 19. However, as the subgroups in the additional columns illustrate, the significant number of small binaries in the binary group artificially brings down the average.

The second through seventh data columns split the binary group into subsets by number of `restores`. The first subgroup with number of `restores` from 9-347 comprises the vast bulk of the overall group with 729 binaries out of 969 total searched. Unsurprisingly, this group has very few matches aside from the 2 trivial gadgets and an average matched result of 2.09.

The other (third through seventh) data columns split the remaining binaries into much smaller subgroups (between 14 and 69 binaries each). Searches by `traitor` on these subgroups yield more matches, both in terms of different gadgets with at least some binary matches, as well as the average number of gadgets matched per subgroup with increases from 3.13 to 5.86 across the subgroups. The minimum number of matched gadgets for a target binary for all subgroups was 2, indicating that (1) some binaries either don't have sufficient code diversity in instructions leading up to `ret`, `restore` sequences, (2) better, more general queries need to be provided to `traitor`, and/or (3) `traitor` has fundamental limitations on certain binaries. On the other side, the maximum number of matches ranged from 7-11 gadgets out of 19 for all subgroups, which comprises a substantial portion of the core gadget set.

One of the core aspects of the return-oriented programming attack is implementing gadgets that together form a Turing-complete set and thus show that the attack is capable of any computation that can be performed by the underlying system. As our original set of 19 gadgets is Turing-complete, and we also check the matches for individual binaries for a Turing-complete subset of the 19 core gadgets (since no binary matched all of them). We use a very rough approximation for assessing Turing-completeness by requiring:

1. all of the memory load, memory store, variable assignment and constant assignment gadgets,

2. either:

   (a) two of the negation, addition, and subtraction gadgets, or

   (b) the 'and' and 'not' gadgets or the 'or' and 'not' gadgets; and

3. one of the three conditional branch gadgets.

We pose that this is a reasonable criteria for a truly "useful" gadget set, as a higher level exploit abstraction (like an API or compiler) could be integrated on top of a binary with these gadgets.[2] Binaries with gadget matches meeting this operational definition are counted in the "Complete" row of Table 7.2. As the table shows, for our untrained search on the first binary group, we were unable to get any matches. Nonetheless, even a subset of a "complete" set would still be useful to an attacker.

Thus, although there are certain gadgets (the not and system call gadgets) that were not matched and we found no "complete" set across the entire binary group, `traitor` was successful in finding a subset of the core gadgets in a good number binaries, and in many cases a sizable number of gadgets within a single binary.

7.3 First Binary Group, Trained

The search pass on the first binary group discussed in Section 7.2 used the default query configuration that was developed against a single binary, the target libc used for our gadget collection in Chapter 5. While our results in Section 7.2 did show `traitor` to be effective in discovering gadgets, a natural follow-on issue is to what degree the search results for the first binary group can be improved by specifically training the gadget search queries.

In this section, we discuss our results for improving the gadget set query on a second pass of the same binary group.

[2] We also note that although the function call gadget is not necessary to this definition, it one of the most often matched gadget in every one of our aggregate search runs.

7.3.1 Methodology

We first analyzed the aggregate results from Section 7.2 and identified the specific gadget queries that seemed to yield few results. We chose several reasonably-sized binaries to use as our retraining code bases, and began modifying and adding specific queries for each core gadget list query.

We tried and tested various modified queries in `traitor`'s interactive mode against our training binaries and original target libc. In most cases, we expanded the existing queries to be broader. We verified that the modifications to our original queries were at least as broad as the original query. We also added anywhere from 0-4 extra gadget queries to each gadget list query in the core search set. And, we modified the core `traitor` search code to add a new query class.

Our analysis and query modifications took about two days to finish. Our custom revisions to the `traitor` search engine were completed in around four hours. Thus, at a high level, the relative ease with which `traitor` enables training queries and even tweaking the underlying search engine demonstrates the overall flexibility and (as our results will show) power of the automated search approach.

7.3.2 Query Modifications and Training

Our preliminary analysis of the first binary group results revealed that many of the gadget queries were more restrictive than need be. The first query aspect we expanded was to transform memory argument queries such as "`%i[0-7]`" that matched registers only (*e.g.*, "`[%i0]`") into "register plus immediate" queries of type `ArgumentPlus` that would additionally match mixed register-intermediate arguments like "`[%i0 + 0x1]`," "`[%i0 + 0xaf12]`," etc. Our analysis showed that a great number of memory accesses are with arguments of this mixed type.

Gadgets that include sequences with mixed register-intermediate arguments for memory instructions have to be encoded to "undo" the addition operations. But, in most cases, this is a simple operation implemented under the hood by an abstraction layer like the previously discussed exploit API or compiler. And, ultimately, some of the gadgets in our original manual libc catalog did require analogous encod-

ings. It was during this phase of our training experiment that we actually modified the `traitor` search engine code base to add the `ArgumentPlus` query class.

Fortunately, the rest of our query training did not require changing the underlying engine. General techniques we used to produce better results included:

- **Pad "Nop" Frames for Output Registers**: Exploit frames generally use the input and local registers for directly encoded registers, as both types are restored from the stack to the register window on an incoming return. However, the output registers are also available in a given stack frame context — they just have to be set by the *previous* frame (as the input register before the window slide). Accordingly, we actually can use output registers for a given exploit frame (in a gadget query) by padding an extra "`nop`" frame before it. Recall that our "`nop`" frames are just a `ret-restore` suffix, and we can simply encode the input registers in the "`nop`" frame, and they will be available as the output registers in the next frame.

- **Use Output Registers Within Sequences**: A similar technique is to use the output registers *within* a sequence. The key is to realize that although we cannot set the output register values to restore from the stack (without padding), many registers in our queries do not need preset values. For instance, if a sequence has a first instruction write to a destination register, then a second instruction stores that same register to memory, the actual preset value of the register is irrelevant. Accordingly, we can permit the argument query in this case to use input, local or output registers.

- **Use Global Registers**: Although all of the gadgets in the catalog in Chapter 5 and queries up until the training phase relied exclusively on input, local, and output registers, our analysis of sample binaries revealed common use of a specific global register, `%g1`. `%g1` is "special" in that as a global register, its value does not change from sequence to sequence as it is outside of the sliding register window. Another way to look at `%g1` is that it is like the basic `eax`, etc. registers on the Intel x86 platform.

We formulated a sequence query to set the `%g1` register, which we then incorporated into several new gadget queries. If the `%g1` set query is successful, then we can expand our queries for critical instruction queries (such as a branch match, etc.) to also include `%g1` as a potentially useful register. We note in passing that a set `%g1` sequence query yielded no results in our original target libc, and hypothesize that the use of `%g1` may be different across compilers, as its use is largely optional versus conventional input, local and output registers.

We also modified and added queries specific to certain gadgets in the course of binary analysis:

- **Or Gadget**: From simple static analysis of some example binaries in the first binary group, we determined that a number of "`or`" instruction sequences occur in the following two forms: (1) `or <reg>, %g1, <reg>`, and (2) `or %g1, <reg>, %g1`, where "`<reg>`" is an ordinary input, local or output register. This realization led us to develop the `%g1` register technique above to use `%g1` for our "`or`" gadget queries to increase the number of or gadget matches.

- **Not Gadget**: The not gadget is quite elusive and our original gadget from libc was quite implementation specific. It is thus not too surprising that our first pass on the first binary group yielded no matches for the not gadget. In our training analysis, we discovered that very few "`not`" instructions actually occur in our target corpus (likely due to the fact `not` is a synthetic instruction on SPARC). Thus, we looked to equivalent operations, and found the most likely candidates to be "`xnor`" and "`andn`". Performing specific static analysis on example binaries in the first target group, we were able to find matches in specific binaries using the `%g1` register technique. Our resulting queries met with some success in both the first and second binary groups (although less than 50 matching binaries in each case).

- **System Call Gadget**: The system call gadget proved to be the most difficult gadget for `traitor`. Both the first and second pass on the first binary group failed to yield any matches. Our research and static assembly analysis revealed

that the specific trap we were searching for, "`ta 0x8`," does not occur in *any* of the binaries in our search group in any context. Moreover, virtually all traps were "`ta 0x3`," which is the flush register window trap, and even those only numbered around 40 total across *all* binaries. Thus, at a fundamental level, the system call gadget was impossible for the 969 gadgets in our first search group and we postulate that it is likely to be rare in other binaries, except for those like libc, which provide the conventional entry point wrappers for system calls.

7.3.3 Results

As will be discussed in more detail in Section 7.3.4, the trained queries performed much better on the first binary group than the previous untrained results presented in Section 7.2.1.

The second pass trained search took just over 11 hours to complete, increasing the total search time for the previous search run by about five hours. However, this is an expected change as the number of specific gadget queries for the core set of 19 gadget lists was increased from 54 in the first pass to 83 in the trained search. Essentially, we increased our likelihood of getting gadget matches by throwing more queries at the target binaries.

For individual target binary searches, our fastest search took 0.16 seconds for the same 9 `restore`s / 122 instruction target from the previous section. Our longest search took 28 minutes for our largest target binary, which contains 30,335 `restore`s / 1,316,921 instructions. The minimum per-instruction search time was 0.0005 seconds, the maximum was 0.0017 seconds, and the average was 0.0007 seconds — fairly comparable to the untrained search pass, and quite reasonable given the increased total number of queries in the second search.

Our results for the second, trained pass on the first binary group are presented in Figure 7.2. Like before, every search produced at least two gadget matches for our trivial gadgets (`nop` and branch always). Figure 7.2 shows a similar correlation to Figure 7.1 in that the number of distinct gadget matches for a target binary increases as the number of `restore`s increases. A quick visual inspection of Figure 7.2

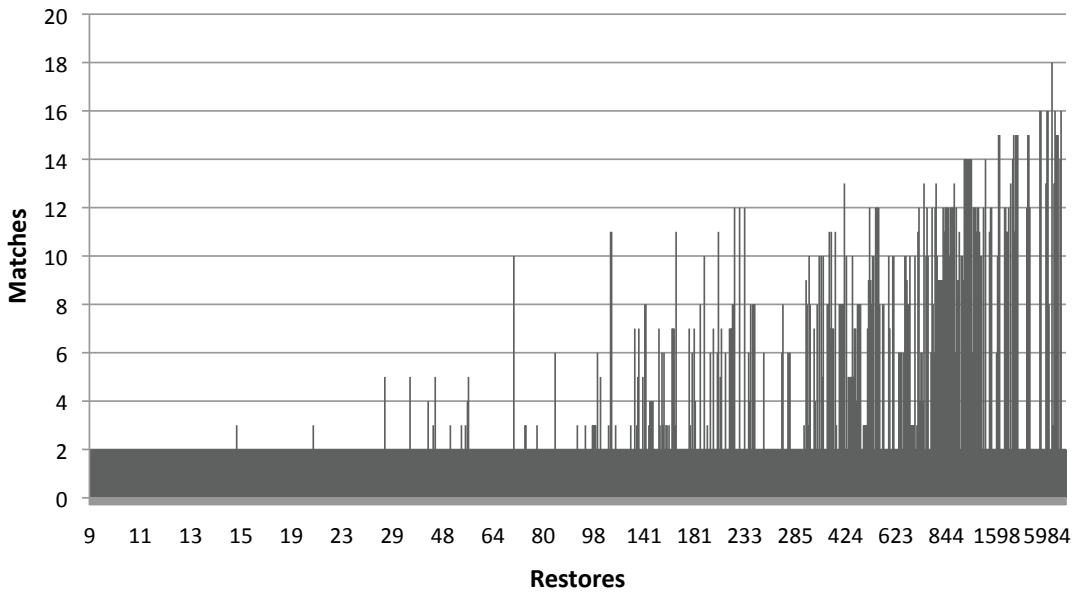

Figure 7.2: Group 1 (Trained) Matches by Number of Restores

reveals that the query training did significantly increase both the number of binaries with more than the trivial two matches and the number of gadgets matched for binaries within the search group. The maximum number of matched gadgets for a single binary was 18 out of 19 in this run. (As discussed previously, there were no matches for the system call gadget for either the untrained or trained search passes on the first binary group). Thus, our trained search shows that the return-oriented exploits we described in Chapters 5-6 are demonstrably not limited to a single target library, as we have now automatically discovered gadget sets in other binaries that are gadget-for-gadget equivalent to the original target libc catalog presented in Chapter 5 (excepting the system call gadget).

Table 7.3 describes our second pass search results in greater detail. The average number of gadget matches for binaries in the entire search group was 3.8 out of 19 total. As noted previously, this average skews downward due to the large number of small binaries in our search group that have few matches.

For the segmented subgroups, the first subgroup (9-347 `restore`s) maintained a very low average of 2.55 gadget matches, while comprising most of the binaries in

Table 7.3: Group 1 (Trained) Matches

Restores Range		**All** (9-30335)		9-347		351-597		614-892		922-2705		2855-7971		10164-30335	
Num. Binaries		**All** (969)		729		69		59		68		30		14	
Complete		27	*3%*	0		2	*3%*	3	*5%*	9	*13%*	8	*27%*	5	*36%*
Min. Matches		2		2		2		2		2		2		2	
Max. Matches		18		12		13		13		15		18		16	
Avg. Matches		3.80		2.55		6.42		7.75		9.00		6.50		8.50	
0	Nop	969	*100%*	729	*100%*	69	*100%*	59	*100%*	68	*100%*	30	*100%*	14	*100%*
1	Memory Load	210	*22%*	62	*9%*	40	*58%*	42	*71%*	48	*71%*	11	*37%*	7	*50%*
2	Memory Store	210	*22%*	62	*9%*	40	*58%*	42	*71%*	48	*71%*	11	*37%*	7	*50%*
3	Assignment	230	*24%*	74	*10%*	46	*67%*	44	*75%*	48	*71%*	11	*37%*	7	*50%*
4	Const. Assn.	184	*19%*	59	*8%*	33	*48%*	39	*66%*	37	*54%*	10	*33%*	6	*43%*
5	Increment	145	*15%*	28	*4%*	31	*45%*	27	*46%*	42	*62%*	10	*33%*	7	*50%*
6	Decrement	83	*9%*	10	*1%*	11	*16%*	16	*27%*	30	*44%*	9	*30%*	7	*50%*
7	Negation	78	*8%*	8	*1%*	12	*17%*	16	*27%*	25	*37%*	10	*33%*	7	*50%*
8	Addition	106	*11%*	12	*2%*	17	*25%*	23	*39%*	38	*56%*	9	*30%*	7	*50%*
9	Subtraction	84	*9%*	10	*1%*	14	*20%*	17	*29%*	26	*38%*	10	*33%*	7	*50%*
10	And	47	*5%*	5	*1%*	3	*4%*	8	*14%*	25	*37%*	5	*17%*	1	*7%*
11	Or	58	*6%*	7	*1%*	5	*7%*	11	*19%*	22	*32%*	8	*27%*	5	*36%*
12	Not	47	*5%*	1	*0%*	4	*6%*	7	*12%*	20	*29%*	8	*27%*	7	*50%*
13	Branch Always	969	*100%*	729	*100%*	69	*100%*	59	*100%*	68	*100%*	30	*100%*	14	*100%*
14	Branch Equal	37	*4%*	2	*0%*	1	*1%*	4	*7%*	14	*21%*	9	*30%*	7	*50%*
15	Branch LTE	4	*0%*	0		0		0		2	*3%*	1	*3%*	1	*7%*
16	Branch LT	9	*1%*	1	*0%*	2	*3%*	0		3	*4%*	2	*7%*	1	*7%*
17	Function Call	214	*22%*	59	*8%*	46	*67%*	43	*73%*	48	*71%*	11	*37%*	7	*50%*
18	System Call	0		0		0		0		0		0		0	

the search group (729 of 969). All other subgroups performed much better, with average gadget matches ranging from 6.42 to 9.0. Thus, for a much larger portion of our search subgroups, we are getting sizable portions of the core gadget set, and for some subgroups, we average half of the core set for the entire subgroup. The maximum number of gadget matches across all subgroups ranged from 12-19, indicating that for any binary size, there are at least some that have the vast majority of the gadget set (and in many cases are just missing two of the three branch gadgets and system call gadget).

The trained query also produced much better qualitative results in terms of our loosely-defined Turing-completeness criteria, with 27 binaries producing "complete" gadget sets. And, looking at the sized subgroups, particularly above the 900 `restore` instruction count, the percentage of complete gadget sets increases from 13% to 36% for the largest binary subgroups.

Thus, `traitor` is likely to find a good number of gadgets for binaries with over 350 `restore`s, and a 25% or better chance of finding a "complete" gadget set for binaries over the 2700 `restore` mark for the specific targets in our first binary group. Ultimately, Figure 7.2 and Table 7.3 demonstrate that for at least this sampling of almost one thousand binaries, the number of `restore`s is directly correlated with the likelihood and magnitude of return-oriented exploits in an arbitrary binary.

7.3.4 Results Comparison

Table 7.4 compares the original search pass to our second (trained) search of the first binary group. The last two table columns present the absolute increase in binaries with gadget matches and percentage improvement for the second search. As Table 7.4 illustrates, running the improved, trained queries on the first binary group had a significantly positive impact on the number of gadget matches. Aside from the two trivial gadgets (that already had 100% matches) and the system call gadget (with no matches), every gadget category in the remaining 16 gadgets had at least a 48% improvement and 14 of those gadgets had 100%+ improvement. In terms of absolute numbers of matches, aside from the system call gadgets and the two "difficult" branch gadgets (branch less-than-or-equal and branch less-than), the

Table 7.4: Group 1 Untrained vs. Trained Matches

Search		Untrained	Trained	*Change* (+)	
Num. Binaries		969	969		
Min. Matches		2	2	0	*0%*
Max. Matches		14	18	4	*29%*
Avg. Matches		2.56	3.80	1.24	*48%*
0	Nop	969	969	0	*0%*
1	Memory Load	38	210	172	*453%*
2	Memory Store	1	210	209	*20900%*
3	Assignment	113	230	117	*104%*
4	Const. Assn.	107	184	77	*72%*
5	Increment	27	145	118	*437%*
6	Decrement	6	83	77	*1283%*
7	Negation	3	78	75	*2500%*
8	Addition	43	106	63	*147%*
9	Subtraction	31	84	53	*171%*
10	And	20	47	27	*135%*
11	Or	17	58	41	*241%*
12	Not	0	47	47	–
13	Branch Always	969	969	0	*0%*
14	Branch Equal	25	37	12	*48%*
15	Branch LTE	2	4	2	*100%*
16	Branch LT	4	9	5	*125%*
17	Function Call	106	214	108	*102%*
18	System Call	0	0	0	*0%*

second trained search pass yielded at least 37 individual binary matches for *every* remaining gadget category in the core set of 19.

All in all, the results from our second search pass demonstrate that: (1) we can effectively retrain and augment the core set of queries to produce significant percentage increases in nearly all gadget categories, (2) the absolute number of second pass matches for almost every gadget is substantial with half of the categories getting at least 100 matched binaries out of 969, and (3) training `traitor` is quick and effective — our improved search results came from only two day's worth of query training and code modifications to the core `traitor` search engine.

7.4 Second Binary Group

Our second binary group is comprised of an additional 220 packages downloaded from the Sun Freeware website [5]. The group contains 863 binaries and has an instruction count roughly similar to the first binary group. For our measurements, we ran the "trained" query from Section 7.3 on all the individual binaries in our second group.

7.4.1 Results

Our search of the 863 binaries had a total run time of around seven and a half hours. Our fastest single binary search was 0.16 seconds for a 6 `restores` / 215 instruction target, and our slowest was 45 minutes for a large 30,314 `restores` / 1,871,604 instruction target. For per-instruction times, the minimum was 0.0005 seconds, the maximum was 0.0014 seconds, and the average was 0.0007 seconds — nearly identical to the per-instruction times for the trained search on the first binary group.

Figure 7.3 shows gadget matches for the second binary group. It appears that the improvements made to our query set in Section 7.3 are generally applicable. The maximum number of gadget matches for a given binary was 18 out of 19, matching the best performing binary search in the first binary group. The density of 3+ gadget matches (across the x-axis) and magnitude of number of matches (across the y-axis)

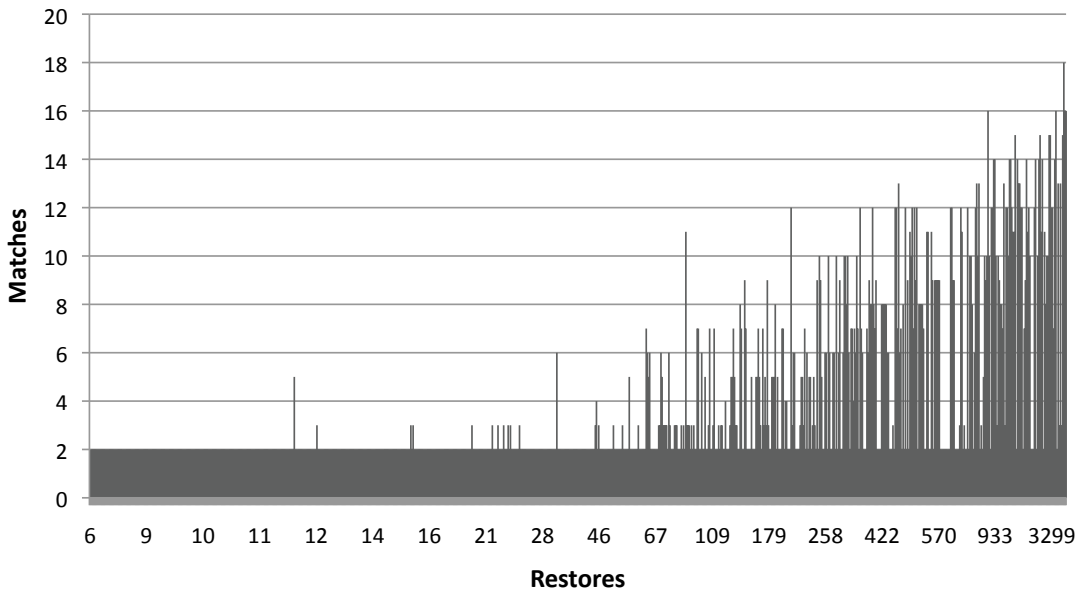

Figure 7.3: Group 2 Matches by Number of Restores

in Figure 7.3 shows a clear relationship between the number of `restore` instructions in a target and the number of gadgets likely to be found.

Table 7.5 describes our search results for the second binary group in detail. The average number of gadget matches for all binaries in the search group was 3.73 out of 19 total, quite similar to the 3.8 average from the first binary group.

The segmented subgroups in the second binary group had a good portion of small binaries, but overall, less binaries in the "large" subgroups when divided along similar `restore` count ranges. Like the first binary group, the first "small" subgroup of 685 targets in our second binary group had low overall and specific gadget match averages. For the larger remaining subgroups, the results were impressive (even if for a smaller sampling in each subgroup). Average gadget matches for the remaining groups increased from 6.27 to 16.25 going from small to large. The maximum number of matches in the remaining subgroups was similarly high — from 13 to 18 out of 19 gadgets total.

We also found a good number of binaries with "complete" sets in the second binary group, with 24 overall sets. For segmented subgroups over 900 `restore`s,

Table 7.5: Group 2 Matches

Restores Range		**All** (6-30314)		6-350		377-598		602-874		933-2747		2812-7977		8880-30314	
Num. Binaries		**All** (863)		685		83		32		48		11		4	
Complete		24	*3%*	0		3	*4%*	2	*6%*	11	*23%*	4	*36%*	4	*100%*
Min. Matches		2		2		2		2		2		2		15	
Max. Matches		18		12		13		16		15		16		18	
Avg. Matches		3.73		2.63		6.27		7.94		9.73		10.00		16.25	
0	Nop	863	*100%*	685	*100%*	83	*100%*	32	*100%*	48	*100%*	11	*100%*	4	*100%*
1	Memory Load	183	*21%*	60	*9%*	50	*60%*	20	*63%*	39	*81%*	10	*91%*	4	*100%*
2	Memory Store	181	*21%*	60	*9%*	50	*60%*	20	*63%*	39	*81%*	8	*73%*	4	*100%*
3	Assignment	216	*25%*	94	*14%*	50	*60%*	21	*66%*	39	*81%*	8	*73%*	4	*100%*
4	Const. Assn.	176	*20%*	65	*9%*	44	*53%*	22	*69%*	35	*73%*	6	*55%*	4	*100%*
5	Increment	104	*12%*	26	*4%*	17	*20%*	17	*53%*	33	*69%*	7	*64%*	4	*100%*
6	Decrement	68	*8%*	6	*1%*	18	*22%*	9	*28%*	26	*54%*	5	*45%*	4	*100%*
7	Negation	59	*7%*	10	*1%*	9	*11%*	14	*44%*	16	*33%*	6	*55%*	4	*100%*
8	Addition	76	*9%*	9	*1%*	20	*24%*	10	*31%*	28	*58%*	5	*45%*	4	*100%*
9	Subtraction	61	*7%*	10	*1%*	10	*12%*	15	*47%*	16	*33%*	6	*55%*	4	*100%*
10	And	31	*4%*	1	*0%*	9	*11%*	7	*22%*	7	*15%*	3	*27%*	4	*100%*
11	Or	61	*7%*	13	*2%*	18	*22%*	6	*19%*	15	*31%*	6	*55%*	3	*75%*
12	Not	34	*4%*	4	*1%*	1	*1%*	5	*16%*	16	*33%*	4	*36%*	4	*100%*
13	Branch Always	863	*100%*	685	*100%*	83	*100%*	32	*100%*	48	*100%*	11	*100%*	4	*100%*
14	Branch Equal	33	*4%*	0		7	*8%*	2	*6%*	16	*33%*	4	*36%*	4	*100%*
15	Branch LTE	10	*1%*	0		1	*1%*	1	*3%*	6	*13%*	1	*9%*	1	*25%*
16	Branch LT	4	*0%*	0		0		1	*3%*	1	*2%*	1	*9%*	1	*25%*
17	Function Call	195	*23%*	74	*11%*	50	*60%*	20	*63%*	39	*81%*	8	*73%*	4	*100%*
18	System Call	0		0		0		0		0		0		0	

`traitor` found complete sets in 23%, 36% and 100% of the binaries in the respective groupings. Thus, although admittedly not an enormous sampling (with only around 100 binaries), considering the "complete" results for the trained query on both the first and second binary groups, there is at least a strong suggestion that `traitor` has a decent likelihood of finding Turing-complete gadget sets in arbitrary binaries at various instruction count thresholds.

Alternately, we can make the softer claim that in light of our three search runs on two binary groups, there is decent evidence that `traitor` can effectively find significant subsets of our core 19 sought gadgets in previously unknown binaries. The results from the second binary group show a definite positive correlation between the likelihood of gadgets and the `restore` count in a given target binary.

7.5 Lessons and Implications for Automated Gadget Searching

The search experiments in this chapter provide us with interesting information about the nature of many of our gadgets. Although general query generalizations were able to improve our search results in most cases, several gadgets required specific analysis, considering issues such as the common case for certain instructions in the target code base, and possibly compiler-dependent instruction generation differences. In all cases except the system call gadget, we were able to broaden our gadget queries after examining example target binaries, and offer that similar binary-specific training may be successful for other targets outside our two search groups.

Ultimately, we were unsuccessful in finding any matches for the system call gadget due to a dearth of usable "`ta`" instructions with the correct trap code. We argue that this probably is rooted in the fact that SPARC binaries to not typically trap to the kernel, instead relying on wrapper functions in libraries like libc (where we found our original system call gadget matches). It may be possible to find a general-purpose system call gadget in the future by looking to trap instructions other than `ta`.

Our experience in running `traitor` on large search sets demonstrates that an

automated approach to return-oriented gadget searches is both quick and practically feasible. For the main binary we trained on in the first binary group, we were able to raise the number of gadget matches from 10 to 16. Using this improved query on the entire binary group, we achieved the improved results discussed in Section 7.3.3. And, the same trained query performed equally well on an unknown binary group, as shown in Section 7.4.1.

Given the speed with which we were able to train the original query and even augment the `traitor` search code, we pose that discovering and implementing full return-oriented programming attacks on new and unknown binaries is not only possible, but increasing likely as the target code size increases. We also point out that searching with `traitor` is fast — our longest search on an extremely large binary only took 45 minutes on a single server-class computer. Together, our results evidence that `traitor` is powerful, general, extensible, and efficient.

Finally, we note that all our experiments in this chapter focus on single binaries in isolation. In reality, an attacker looking to implement a return-oriented exploit can not only use the core code of a vulnerable program, but any dynamically loaded libraries, such as libc, as well. Considering all the "good" code that is loaded by a typical program at runtime, we conclude with the observation that combining available code bases for return-oriented attacks makes it very likely that a "complete" set, or at least a useful subset, of gadgets will be found in any arbitrary program

Chapter 8

Conclusion

Like many historically widespread vulnerabilities and exploits that were initially thought to be limited in pervasiveness or applicability, we offer that return-oriented programming is evolving from its original specific and singular applications to a generally applicable and practical threat across computer systems believed to be protected by W⊕X.

In this thesis, we have described porting the return-oriented model from the x86 to the radically different SPARC architecture. Our resulting SPARC gadget set for our target Solaris libc binary is Turing-complete on inspection and eminently capable of use in practical and extensible attacks [4].

More importantly, we have introduced a tool, `traitor`, which automates the previously painstaking task of performing static assembly code analysis on target binaries to discover usable return-oriented gadgets. `traitor` provides an efficient search algorithm and extensible query language to permit users to find Turing-complete gadget sets in previously unknown binaries in minutes instead of weeks or months.

Our experiences running `traitor` on many target binaries have provided much insight into the future directions and impacts of return-oriented programming generally. First, our results for searching nearly 2,000 binaries empirically demonstrates that `traitor` can find full or partially Turing-complete gadget sets for a substantial portion of the target search set. Second, `traitor` can be quickly tuned to specific

binaries either through query modifications and/or enhancing the underlying search algorithm code and produce significantly improved search results. Finally, looking to our results broken down across instruction count thresholds, we see good experimental support for the return-oriented thesis — as target code size increases, so do the chances for discovering a usable and/or Turing-complete gadget set in an arbitrary binary. Moreover, we point out that our measurements looked at isolated binaries without considering dynamic libraries loaded. In the wild, an attacker focusing on a specific binary can not only use the core binary code, but all dynamically loaded libraries (*e.g.*, libc), which we argue makes the likelihood of finding a full or useful-enough return-oriented gadget set in an arbitrary binary very good.

Against the backdrop, we pose that the return-oriented programming model is powerful and extensible, and with our present research, efficiently and generally applicable to wide array of programs (that already have some entry vector vulnerability). We offer that it is time to reconsider the fundamental notions of the W$\oplus$X defense model as automated tools dramatically increase the ease with which "bad" computations can be performed solely with "good" code in vulnerable programs and empirical data provides a glimpse of the true and possibly pervasive extent of return-oriented exploits in programs in the wild.

Bibliography

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

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

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

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, Oct. 2008.

[5] S. Christensen. Sunfreeware.com. `http://www.sunfreeware.com/`.

[6] M. Conover. w00w00 on heap overflows, 1999. `http://www.w00w00.org/files/articles/heaptut.txt`.

[7] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo*, volume 02, page 1119, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

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

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

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

[11] 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. http://support.microsoft.com/KB/875352.

[12] A. Noordergraaf and KeithWatson. Solaris™ operating environment security, Jan. 2000.

[13] OpenBSD Foundation. OpenBSD 3.3 release, May 2003. http://www.openbsd.org/33.html.

[14] OpenBSD Foundation. OpenBSD 3.4 release, Nov. 2003. http://www.openbsd.org/34.html.

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

[16] PaX Team. Homepage of the PaX Team. http://pax.grsecurity.net/.

[17] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[18] 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.

[19] Solar Designer. StackPatch. http://www.openwall.com/linux.

[20] Solar Designer. Getting around non-executable stack (and fix). Bugtraq, Aug. 1997.

[21] SPARC Int'l, Inc., Englewood Cliffs, NJ, USA. *The SPARC Architecture Manual (Version 9)*, 1994.

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