

# **IMMUNIZING BINARY EXECUTABLES AGAINST RETURN-ORIENTED PROGRAMMING**

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Kaan Onarhoğlu

July, 2010

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

---

Assist. Prof. Dr. Ali Aydın Selçuk (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

---

Assist. Prof. Dr. İbrahim Körpeoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

---

Assoc. Prof. Dr. Mustafa Akgül

Approved for the Institute of Engineering and Science:

---

Prof. Dr. Levent Onural  
Director of the Institute

# ABSTRACT

## IMMUNIZING BINARY EXECUTABLES AGAINST RETURN-ORIENTED PROGRAMMING

Kaan Onarlioğlu  
M.S. in Computer Engineering  
Supervisor: Assist. Prof. Dr. Ali Aydın Selçuk  
July, 2010

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia.

ROP attacks utilize short code sequences each ending with a *free-branch* instruction, i.e. an instruction that allows the attacker to control the execution flow. Identifying such sequences, or *gadgets*, available in binary executables and chaining them together, it is possible to perform arbitrary computations. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

In this work, we present a compiler-based approach that represents the first practical solution against any possible form of ROP. Our solution is able to protect the aligned free-branch instructions to prevent them from being misused by an attacker, and to eliminate all unaligned free-branch instructions inside a binary executable. We developed a prototype based on our approach for the x86 architecture, and evaluated it by compiling GNU `libc` and a number of real-world applications. The results of the experiments demonstrate that our solution is able to prevent any form of return-oriented programming attack.

*Keywords:* Return-oriented programming, return-to-libc, memory corruption vulnerabilities.

## ÖZET

# İKİLİ YÜRÜTÜLEBİLİRLERİN DÖNÜŞE DAYALI PROGRAMLAMAYA KARŞI BAĞIŞIKLANDIRILMASI

Kaan Onarlıoğlu

Bilgisayar Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Yrd. Doç. Dr. Ali Aydın Selçuk

Temmuz, 2010

Modern işletim sistemlerinde bulunan birçok önlem ve koruma mekanizmasına rağmen, bellek bozma açıklarının istismarı yazılım sistemlerinin ve bilgisayar ağlarının güvenliği için hala ciddi bir tehdit oluşturmaktadır. Yakın geçmişte ortaya atılan “Dönüşe Dayalı Programlama (DDP)” isimli istismar tekniği son zamanlarda akademik ortamda oldukça dikkat çekti.

DDP saldırıları, *serbest-dal* yönergeleriyle, yani bir saldırganın yürütme akışını kontrol etmesine olanak sağlayan yönergelerle sonlanan kısa kod dizilerinden faydalanır. İkili yürütülebilirlerde mevcut olan bu tür dizileri, diğer bir deyişle *aygıtları*, teşhis ederek ve bunları birbirlerine zincirleyerek keyfi hesaplamalar yapmak mümkündür. Geçmişte bu konu üzerinde yapılan araştırmalar çoğunlukla orijinal saldırı tekniklerinin geliştirilmesi veya sadece belirli saldırı türevlerini hedef alan kısmi çözümler önerilmesi üzerine odaklanmıştır.

Bu çalışmada, DDP’nin mümkün olan her şekline yönelik ilk pratik çözümü temsil eden, derleyici tabanlı bir yaklaşım sunuyoruz. Çözümümüz, bir saldırganın kötüye kullanmasını engellemek amacıyla hizalı *serbest-dal* yönergelerini koruyabilmekte ve ikili yürütülebilirlerin içerisindeki tüm hizasız *serbest-dal* yönergelerini ortadan kaldıracıdır. Yaklaşımımıza dayanarak x86 mimarisini hedefleyen bir prototip geliştirdik; GNU `libc` ve birkaç gerçek uygulama derleyerek bu prototipi değerlendirdik. Deney sonuçlarına göre, sunduğumuz çözüm her türlü DDP saldırısını engelleyebilmektedir.

*Anahtar sözcükler:* Dönüşe-dayalı programlama, return-to-libc, bellek bozma açıkları.

## **Acknowledgement**

I would first like to thank my supervisor Assist. Prof. Dr. Ali Aydın Selçuk who encouraged me to pursue my master's studies in the field of computer security and lended me his support and guidance throughout my research.

I also gratefully thank Assist. Prof. Dr. Engin Kirda for mentoring me during my visit to Institute Eurécom, where this thesis study was carried out in part; without his supervision this work would not have been possible.

I owe special thanks to Assist. Prof. Dr. Davide Balzarotti for his invaluable contributions that improved this work immensely.

I would also like to thank Leyla, Andrea and all the other great people in Eurécom for their assistance and friendship.

Lastly, I thank The Scientific and Technological Research Council of Turkey for financing my master's studies through the “National Scholarship Programme for MSc Students”.

# Contents

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| <b>2</b> | <b>Memory Corruption Attacks &amp; Defenses</b>   | <b>5</b>  |
| 2.1      | Traditional Attack Techniques . . . . .           | 5         |
| 2.2      | Non-Executable Memory Protection . . . . .        | 6         |
| 2.3      | Return-to-Lib(c) Attacks . . . . .                | 7         |
| 2.4      | Proposed Defense Techniques . . . . .             | 8         |
| <b>3</b> | <b>Return-Oriented Programming</b>                | <b>9</b>  |
| 3.1      | Programming with Gadgets . . . . .                | 10        |
| 3.2      | Gadget Construction . . . . .                     | 15        |
| 3.3      | Proposed Defense Techniques against ROP . . . . . | 17        |
| <b>4</b> | <b>Our Approach: Code without Gadgets</b>         | <b>19</b> |
| 4.1      | Free Branch Protection . . . . .                  | 20        |

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| 4.1.1    | Alignment Sleds . . . . .                                         | 21        |
| 4.1.2    | Return Address Protection . . . . .                               | 22        |
| 4.1.3    | Frame Cookies . . . . .                                           | 23        |
| 4.2      | Code Rewriting . . . . .                                          | 24        |
| 4.2.1    | Register Reallocation . . . . .                                   | 24        |
| 4.2.2    | Instruction Transformations . . . . .                             | 26        |
| 4.2.3    | Jump Offset Adjustments . . . . .                                 | 27        |
| 4.2.4    | Immediate and Displacement Reconstructions . . . . .              | 28        |
| 4.2.5    | Inter-Instruction Barriers . . . . .                              | 28        |
| 4.3      | Limitations of the Approach . . . . .                             | 29        |
| <b>5</b> | <b>Implementation</b>                                             | <b>30</b> |
| 5.1      | Assembly Code Pre-Processor & Binary Analyzer . . . . .           | 31        |
| 5.2      | Obtaining Random Keys . . . . .                                   | 32        |
| 5.3      | Stack Reference Adjustments . . . . .                             | 32        |
| 5.4      | Conditional Code Rewriting . . . . .                              | 34        |
| 5.5      | Return Address and Indirect Jump/Call Protection Blocks . . . . . | 35        |
| 5.6      | Compiling <code>glibc</code> . . . . .                            | 37        |
| 5.6.1    | Multiple Entry Points . . . . .                                   | 38        |
| 5.6.2    | Functions that Access the Saved Return Address . . . . .          | 38        |

|          |                                                             |           |
|----------|-------------------------------------------------------------|-----------|
| 5.6.3    | Jumps between Functions . . . . .                           | 39        |
| 5.6.4    | Position Independent Code (PIC) Addressing Issues . . . . . | 39        |
| 5.7      | Limitations of the Implementation . . . . .                 | 40        |
| <b>6</b> | <b>Evaluation</b>                                           | <b>42</b> |
| 6.1      | Compilation Results . . . . .                               | 42        |
| 6.2      | Performance Measurements . . . . .                          | 44        |
| <b>7</b> | <b>Conclusions</b>                                          | <b>48</b> |

# List of Figures

|     |                                                                                                                     |    |
|-----|---------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | Stack frame demonstrating how to perform a simple subtraction operation using return-oriented programming . . . . . | 11 |
| 3.2 | Stack frame demonstrating how to load an immediate value into memory using return-oriented programming . . . . .    | 12 |
| 3.3 | Examples of different gadgets that can be extracted from a real byte sequence . . . . .                             | 16 |
| 4.1 | Application of an alignment sled to prevent executing an unaligned <code>ret (0xc3)</code> instruction . . . . .    | 22 |
| 5.1 | Stack frame before and after inserting a cookie . . . . .                                                           | 33 |
| 5.2 | Code inserted to protect the aligned return instructions . . . . .                                                  | 35 |
| 5.3 | Code inserted to protect the aligned indirect jump/call instructions . . . . .                                      | 35 |
| 5.4 | Gadgets available in the return address protection block . . . . .                                                  | 37 |
| 5.5 | Gadget available in the indirect jump/call protection block . . . . .                                               | 38 |

# List of Tables

|     |                                                                                                                                           |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------|----|
| 4.1 | ModR/M values encoding <code>ret</code> opcodes . . . . .                                                                                 | 25 |
| 4.2 | SIB values encoding <code>ret</code> opcodes . . . . .                                                                                    | 25 |
| 6.1 | Statistics on binaries compiled with G-Free (RAP=Return Address Protection, JCP=indirect Jump/Call Protection) . . . . .                  | 43 |
| 6.2 | The increase in size for binaries compiled with G-Free . . . . .                                                                          | 44 |
| 6.3 | Performance comparisons of common Linux applications when the application and all the linked libraries are compiled with G-Free . . . . . | 45 |
| 6.4 | Performance comparison of the original and G-Free <code>glibc</code> using benchmarks from the Phoronix Test Suite . . . . .              | 47 |

# Chapter 1

## Introduction

As the popularity of the Internet increases, so does the number of attacks against vulnerable services [5]. A common way to compromise an application is exploiting memory corruption vulnerabilities in order to divert the program execution to a location under the control of the attacker. In this kind of attacks, the first step is to overwrite a function pointer or a value which will eventually be copied to the instruction pointer (e.g., the saved return address) in memory. Overflowing a buffer on the stack [7] or exploiting a format string vulnerability [28] are well-known techniques used to this effect. Once the attacker is able to hijack the control flow of the application, the next step is taking control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks and to increase the complexity of performing these two attack steps [13, 20, 12, 14, 38]. In particular, all modern operating systems support some form of memory protection mechanism [35] to prevent programs from executing code that resides in certain memory regions. Protection mechanisms could be emulated in software by the operating system or implemented in hardware using the *No eXecute* bit for the CPUs that support it. Regardless of the technology, the goal is to protect against code

injection attacks by setting the permissions of the memory pages that contain data (such as the stack and the heap of the process) as non-executable.

One of the techniques to bypass non-executable memory involves reusing the functionality provided by the exploited application as opposed to relying on injected code. Using this technique, which was originally called return-to-lib(c) [33], an attacker can prepare a fake frame on the stack and then transfer the program execution to the beginning of a library function. Since some popular libraries (such as `libc`) contain a wide range of functionality, this technique is still sufficient to take control of the program (e.g., by exploiting the `system` function to execute `/bin/sh` and spawn a shell).

In 2007, Shacham [31] introduced an evolution of the return-to-lib(c) [33] attack called Return-Oriented Programming (ROP). The main contribution of ROP is to show that it is possible for an attacker to perform arbitrary computations and achieve Turing completeness without injecting any new code inside the application. Shacham's work disproved the wrong intuition that preventing the injection of malicious code is enough to prevent, or at least contain, *malicious computations*.

The idea behind ROP is simple: Instead of jumping to the beginning of a library function, the attacker chains together sequences of instructions ending with a return instruction (i.e., `ret`), called gadgets, that have been previously identified inside existing code. The large availability of gadgets in common libraries allows the attacker to implement the same functionality in many different ways. Thus, removing potentially dangerous functions (e.g., `system`) from common libraries is ineffective against ROP and does not provide any additional security.

ROP is particularly appealing for rootkit development since it can defeat traditional defense techniques based on kernel data integrity [39] or code verification [26, 30]. Another interesting domain is related to exploiting architectures with immutable memory protection, for example, to compromise electronic voting machines as shown in [9]. This new technique also attracted the attention of the underground community and in 2010 we observed the first attacks in the wild using ROP to bypass Windows' Data Execution Prevention (DEP) technology [4].

The great interest around ROP quickly evolved into an arms race between researchers. On the one side, the basic technique was refined and extended [8, 10, 36, 17]. On the other side, ad-hoc detection and protection mechanisms to mitigate the attack were proposed [18, 23, 15, 11]. To date, existing solutions have focused only on the basic attack, by detecting, for instance, the anomalous frequency of return instructions executed [11, 18], or by removing the `ret` opcode from binary executables to prevent gadget creation [22]. Unfortunately, a recent advancement in ROP [10] has already raised the bar by adopting different instructions to chain the gadgets together, thus making all existing protection techniques ineffective.

In this work, we generalize from all the details that are specific to a particular exploitation technique to undermine the foundation on top of which return-oriented programming is built: the availability of instruction sequences that can be reused by an attacker. We present a general approach for the x86 architecture that combines different techniques to eliminate all possible sources of reusable instructions. Precisely, we introduce a novel protection technique to prevent the attacker from misusing existing return or indirect jump/call instructions, and we use code rewriting techniques to remove all unaligned instructions that can be used to link the gadgets.

We implemented our solution under Linux as a pre-processor for the popular GNU Assembler. We then evaluated our tool on different real-world applications, with a special focus on the GNU `libc` (`glibc`) library. Our experiments show that our solution can be applied to complex programs and it is able to remove all possible gadgets independent of the mechanism used to connect them together. A program compiled with our system is on average 26% larger and 3% slower (in the case that all the linked libraries are also compiled with our solution). This is a reasonable overhead that is in line with existing stack protection mechanisms such as StackGuard [13].

This work makes the following contributions:

- We present a novel approach to prevent an attacker from reusing fragments of existing code as basic blocks to compose malicious functionality.

- To the best of our knowledge, we are the first to propose a general solution to defeat all forms of ROP. That is, our solution can defend against both known variations and future evolutions of the attack.
- We developed *G-Free*, a proof-of-concept implementation targeting the x86 architecture to generate programs that are hardened against return-oriented programming. Our solution requires no modification to the application source code and can also be applied to system applications that contain large sections of assembly code.
- We evaluated our technique by compiling gadget-free versions of `glibc` and other real-world applications.

The rest of this thesis study is structured as follows: In Chapter 2, we provide a brief insight into traditional memory corruption vulnerabilities. In Chapter 3, we analyze the key concepts of return-oriented programming and identify the core technique that is shared by all the possible attack variations. In Chapter 4, we present our approach for compiling gadget-free applications. In Chapter 5, we describe the details of our prototype implementation. In Chapter 6, we show the results of the experiments we conducted for evaluating the impact and performance of our system. Finally, in Chapter 7, we briefly conclude the thesis.

## **Chapter 2**

# **Memory Corruption Attacks & Defenses**

In this chapter, we provide a summary of the traditional memory corruption attacks that paved the way for return-oriented programming, together with various defense techniques proposed so far.

### **2.1 Traditional Attack Techniques**

Buffer overflows are one of the most common ways to mount memory corruption attacks. A buffer overflow occurs when a program overruns the bounds of a buffer writing data to it and corrupts adjacent memory locations. This is usually caused by programming errors such as improper bounds checking. If the written data is supplied from user input, then an attacker can exploit buffer overflow vulnerabilities to overwrite pointers and data structures in the memory, or to inject malicious code into the process. As a result, buffer overflows are the basis for many memory corruption attacks.

Two well-known forms of buffer overflows are stack-based overflows [7] and heap-based overflows [24]. In a stack-based overflow, the attacker overflows a buffer on the stack to overwrite the saved return address or a function pointer, in order to hijack the execution control flow. In this way, when the executing function returns or the modified function pointer is copied to the instruction pointer, execution jumps to a memory location specified by the attacker. This memory location, which could also be on the stack, typically contains malicious code injected by the attacker prior to or during the buffer overflow. In a heap-based overflow, the overflowed buffer resides on the heap, which is dynamically allocated at runtime by the program. This attack usually takes the form of altering the data structures and pointers used internally by the heap allocation algorithm to hijack the control flow and execute injected code in a similar manner to stack-based overflows.

Format string attacks [28] provide another technique for corrupting the memory. A format string vulnerability occurs when user input is directly used as the format string parameter in a function that performs string formatting, such as `printf`. With the ability to specify any format string, the attacker can read memory (e.g., using the `%x` format token) in order to identify the addresses storing critical values such as the saved return address, and overwrite these addresses (e.g., using the `%n` format token) with pointers to injected malicious code.

## 2.2 Non-Executable Memory Protection

Non-executable memory protection schemes aim to prevent execution of injected code by marking those memory pages intended to store data as non-executable. In this way, attempting to execute injected code on the stack or on the heap causes an exception, averting many of the traditional memory corruption attacks.

Full non-executable memory protection requires support from the CPU, usually called the NX (No-eXecute) bit technology, as well as from the operating system. Some implementations also provide a limited software emulation of the protection if

hardware support is not available.

Linux systems make non-executable memory protection available through PaX [35] and ExecShield [37] patches. Microsoft provides an implementation called Data Execution Prevention (DEP) since Windows XP SP2 and Windows Server 2003 SP1 [2]. OpenBSD offers a similar technology, named  $W\oplus X$ , which enforces a memory region to be marked either as writable or executable, but not both [1]. Mac OS X, FreeBSD, NetBSD and Solaris also have NX bit support for various processor architectures.

## 2.3 Return-to-Lib(c) Attacks

Return-to-lib(c) is an attack technique where the attacker overwrites a return address or function pointer using techniques similar to those discussed in the previous sections, and diverts the control flow of the program to the beginning of a legitimate function [33]. In this way, it is possible to exploit the functionality readily provided by a program to mount an attack. The attacker can also create a fake frame on the stack in order to control the arguments passed to the called function or execute a number of different functions one after another [25].

Since there is no code injection involved in this process, non-executable memory protection schemes are ineffective at securing programs against return-to-lib(c) attacks.

Although it is possible to call any function inside the program or return to any library using this technique, `libc`, which is the shared library providing the C runtime environment in Unix-like systems, is a very popular target. This is because almost every binary executable in Unix-like systems link to `libc`, making it possible for an attacker to construct generic exploits. Moreover, `libc` provides a number of low-level and useful functions which facilitate the attacker's intent to take over the system; for instance, the `libc` function `system` would spawn a shell when invoked with the argument `"/bin/sh"`.

## 2.4 Proposed Defense Techniques

Several defense mechanisms attempt to detect memory corruption exploits which represent a fundamental basic block for mounting return-to-lib(c) attacks.

Stack-Guard [13] is a compile-time solution that aims to detect stack-based overflows by monitoring the integrity of the stack. It instruments each function with a prologue responsible for placing a canary value between a function's local variables and its return address, and an epilogue for validating the integrity of the canary. Consequently, simple stack-based overflow attempts are caught since the overflowed data also overwrites and invalidates the canary. ProPolice [20] extends StackGuard by reordering function parameters and local variables, in order to place them after the pointers that could possibly be overwritten otherwise. PointGuard [12] encrypts pointers stored in memory to prevent them from being corrupted. StackShield [38] uses a shadow return address stack in where the saved return addresses are duplicated; then the duplicate is compared to the original at function exits to see whether they have been tampered with. Similarly, StackGhost [19] presents a shadow return address stack for the SPARC architecture. A complete survey of traditional mitigation techniques together with their drawbacks is presented in [14].

One of the most effective techniques that hamper return-to-lib(c) attacks is Address Space Layout Randomization (ASLR) [34]. In its general form, ASLR randomizes positions of stack, heap, and code segments together with the base addresses of dynamic libraries inside the address space of a process. Consequently, an attacker is forced to correctly guess the positions where these data structures are located to be able to mount a successful attack. Despite the better protection offered by this mechanism, researchers showed that the limited entropy provided by known ASLR implementations can be evaded either by performing a brute-force attack on 32-bit architectures [32] or by exploiting Global Address Table and de-randomizing the addresses of target functions [27].

## Chapter 3

# Return-Oriented Programming

Before presenting the details of our approach, we would like to establish a more precise and general model for the class of attacks we wish to prevent. Therefore, we generalize the concept of return-oriented programming by abstracting away from all the details that are specific to a particular attack technique.

Return-oriented programming is a generalization of known return-to-lib(c) attacks [33, 25, 29] into a more sophisticated and powerful technique. The core idea of ROP is “borrowing” sequences of instructions from existing code (either inside the application or in the linked libraries) and chaining them together in an order chosen by the attacker. Therefore, in order to use this technique, the attacker has to first identify a collection of useful instruction sequences that she can later reuse as basic blocks to compose the code to be executed. A crucial factor that differentiates return-oriented programming from simpler forms of code reuse (such as traditional return-to-lib(c) attacks) is that the collection of code snippets must provide a comprehensive set of functionalities that allows the attacker to achieve Turing completeness *without* injecting any code [31]. The second step of return-oriented programming involves devising a mechanism to manipulate the control flow in order to chain these different code snippets together, and build meaningful algorithms.

Note that these two requirements are not independent: To allow the manipulation of the control flow, the instruction sequences must exhibit certain characteristics that impose constraints on the way they are chosen. For example, sequences may have to terminate with a return instruction, or they may have to preserve the content of a certain CPU register. In this work, we use the term *Gadget* to refer to any valid sequence of instructions that satisfies the control flow requirements. We must point out that, we define a gadget in a slightly different way from the rest of the literature: Whereas past work refers to a combination of many short instruction sequences that perform a *useful task* (e.g., an addition operation) as a gadget, we say each of these sequences are *separate* gadgets (Of course, they can still be chained together when needed). In this way, we strive to refrain from making restrictive assumptions imposed by possible definitions of what a *useful task* is, and thus we make our discussion as general as possible.

### 3.1 Programming with Gadgets

IA-32 instruction set provides a pair of instructions, `ret` and `call`, to facilitate function calls. When `call` is executed, the CPU first pushes the address of the next instruction, the return address, onto the stack and then jumps to the specified target, which is usually the beginning of the called function. After the called function performs its task, it executes the `ret` instruction, which pops the previously saved return address from the stack and jumps to it, back to the callee.

In the most basic form of the ROP attack, attackers exploit this property of `ret` instructions which makes it possible to implicitly jump to an address stored on top of the stack. As a consequence, Shacham initially defined gadgets as useful snippets of code that terminate with a `ret` instruction [31]. The desired control flow is achieved by placing the addresses of these gadgets on the stack and then exploiting the `ret` instructions at the end of each gadget to fetch and copy the next address from the stack to the instruction pointer. In other words, if we consider each gadget as a monolithic

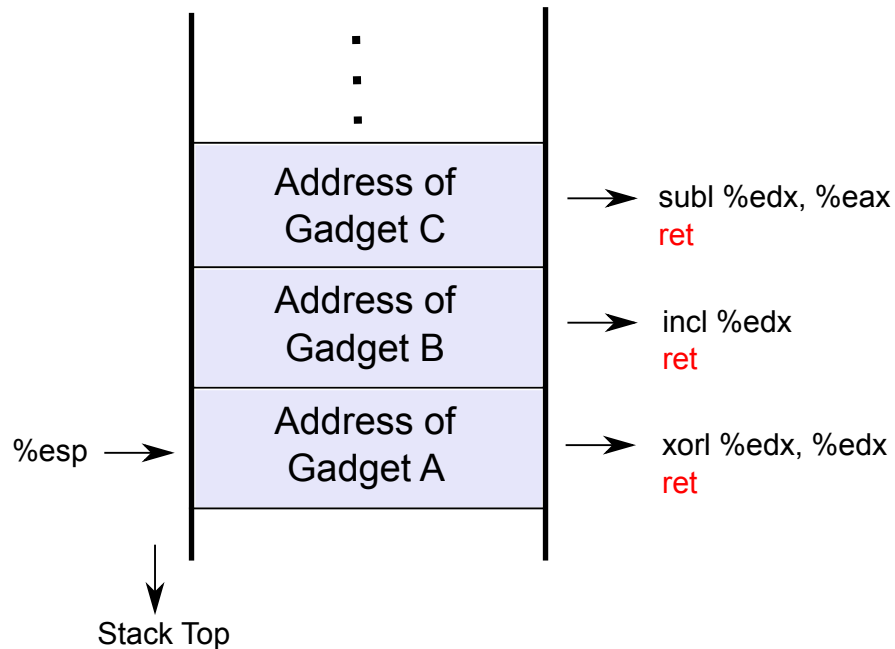


Figure 3.1: Stack frame demonstrating how to perform a simple subtraction operation using return-oriented programming

instruction, the stack pointer plays the role of the instruction pointer in a normal program, transferring control flow from one gadget to the next.

Figure 3.1 demonstrates the described process with a simple example task: subtracting “1” from the value inside the register `%eax`. In order to bootstrap ROP, the attacker crafts the fake stack frame containing the addresses of “Gadget A”, “Gadget B” and “Gadget C” using a stack corruption vulnerability. Afterwards, when the program executes a `ret`, the address of “Gadget A” is popped and copied to the instruction pointer, and return-oriented execution starts. The first instruction to be executed, “`xorl %edx, %edx`”, sets `%edx` to zero. Then next instruction of the current gadget, `ret`, executes, popping and copying the value at the top of the stack (the address of “Gadget B”) into the instruction pointer; and thus chaining the two gadgets. In a similar fashion, “Gadget B” first executes “`incl %edx`”, setting `%edx` to “1”, and then the following `ret` copies the address of “Gadget C” into the instruction pointer. “Gadget C” performs the final step, “`subl %edx, %eax`”, decrementing `%eax` by one, and finally executes a `ret` to chain to the following gadget if there is any.

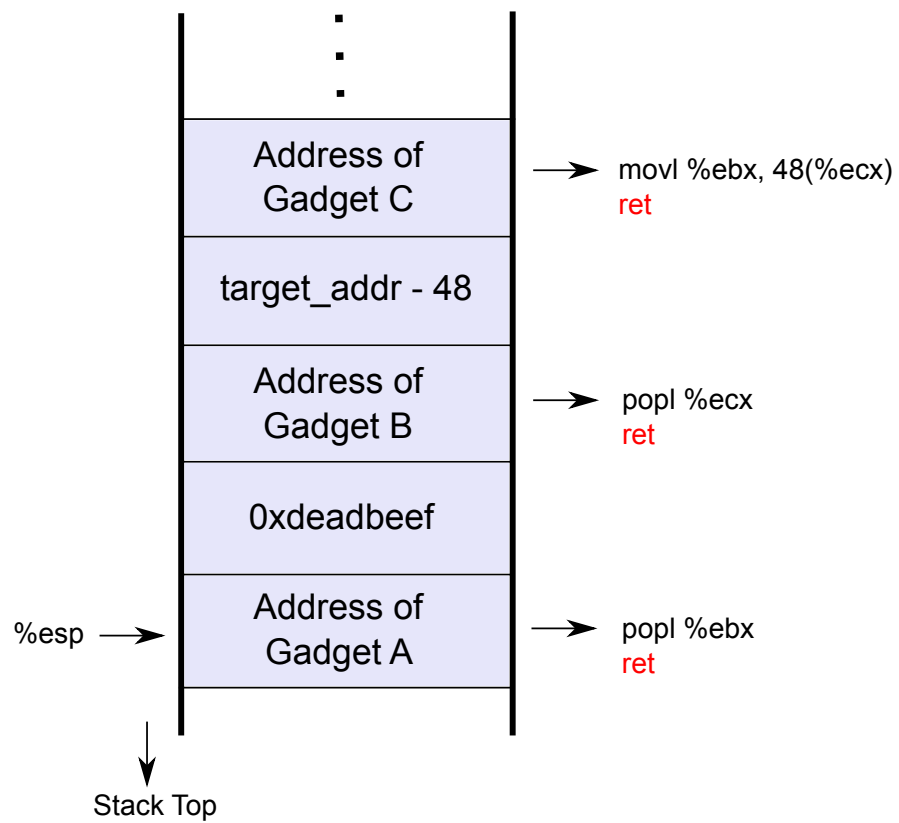


Figure 3.2: Stack frame demonstrating how to load an immediate value into memory using return-oriented programming

The attacker need not have only gadget addresses in the crafted stack, but may also use stack locations to store immediate values, memory operands and temporary values. Figure 3.2 shows how to load the immediate value `0xdeadbeef` into a certain memory location `target_addr`. “Gadget A” pops the top of the stack, which is the immediate value specified by the attacker and stores it in `%ebx`. Now that the top of the stack contains the address of “Gadget B”, the following `ret` successfully chains to the next gadget. “Gadget B” pops the next value into `%ecx`, which is our target memory location minus 48. This is because, “Gadget C” can only write to the memory address `%ecx+48`<sup>1</sup>. The following `ret` chains to the final gadget, “Gadget C”, which loads `0xdeadbeef` to the memory at the address `target_addr`.

Shacham [31] showed that, using these simple chaining mechanics, it is possible to find sufficient gadgets inside the `libc` binary distributed as part of Fedora Core Release 4 to form a Turing complete gadget catalog for the x86 architecture. In order to highlight the significance of this attack, researchers proposed variants of ROP that are suitable for various processor architectures such as SPARC, PowerPC, ARM, Atmel AVR and Z80 [8, 16, 36, 17, 9]. Hund et al. [21] proposed a system that can automatically extract gadgets from kernel code and translate any program written in their dedicated language to return-oriented programs; they have also demonstrated the feasibility of mounting ROP attacks at the kernel level [21].

The use of `ret` instructions is just one possible way of chaining gadgets together. In a recent refinement of the technique [10], Checkoway and Shacham propose a variant of ROP in which *return-like* instructions are employed to fetch the addresses from the stack. A return-like instruction or instruction sequence do not actually contain any `ret` operations, but carry similar semantics. For an example, consider the instruction sequence “`popl %eax; jmp *%eax`”, which behaves in the exact same way as a `ret`: pop an address from the stack and jump to it. In fact, any `jmp` instruction could be a return-like instruction as long as the jump destination is a `ret` (or return-like) instruction.

---

<sup>1</sup>Such restrictions are also common in extracted real-world gadgets since we may not find a gadget for our every specific need; regardless, it is possible to construct the desired semantics by using a number of different gadgets or by carefully crafting operands as shown in this example.

In their work, Checkoway and Shacham show that it is still possible to find a Turing complete gadget collection in `libc` without utilizing any `ret` instructions. However, because “`popl %reg; jmp *%reg`” style sequences are quite rare in regular binaries, indirect jumps, specifically `jmp *(%edx)` instructions, are used as gadget terminators, where `%edx` points to a memory location storing the address of a previously identified return-like sequence.

In practice, it is possible to use any return-like sequence mixed with regular `ret` instructions to extract even larger gadget collections. Although there are no practical examples yet, in theory, it is even possible to exploit other non-return-like instructions such as `call *%reg`, for instance, to directly jump between gadgets without a return-like linking mechanism, or to design control flow manipulation techniques that are not stack-based, but that store values in other memory areas accessible at runtime by an attacker (e.g., on the heap or in global variables).

As a result, in order to find a general solution to the ROP threat, we need to identify a property that all possible variants of return-oriented programming have in common. Kornau [36] identified such a property in the fact that every gadget, in order to be reusable, has to end with a “*free-branch*” instruction, i.e., an instruction that can change the program control flow to a destination that is (or that can be under certain circumstances) controlled by the attacker. According to this definition, in each gadget, we can recognize two parts: the *code section* that implements the gadget’s functionality and the *linking section* that contains the instructions used for transferring the control flow to the next gadget. We can identify the following types of free-branch instructions in the IA-32 instruction set:

- **Return Instructions (`ret`):** There are four different types of `ret` instructions encoded by the opcodes `0xc3`, `0xc2`, `0xcb`, and `0xca`. These opcodes respectively encode a regular return instruction, a return with stack unwinding, a far return (return into a different memory segment), and a far return with stack unwinding. They are regarded as free-branch instructions since an attacker can specify the return destination by injecting values into the stack. Although, the

latter three are slightly harder to use in an actual attack, it is still possible to utilize them by crafting the stack frame carefully.

- **Indirect Jump Instructions (`jmp *%reg, jmp *offset(%reg)`):** Indirect jumps can either jump to an address specified in a register or at a memory location. For the first case, an attacker can assign a target of choice to the register operand using other gadgets; for the second, she can use a memory corruption exploit or again utilize other gadgets to manipulate the memory. Therefore, such instructions are valid free branches. Indirect jumps have already been shown to be useful as return-like instructions, as we have previously discussed in this section; but they can also be used for jumping directly between gadgets.
- **Indirect Call Instructions (`call *%reg, call *offset(%reg)`):** In the exact same way as indirect jumps, destination of indirect calls can be manipulated by an attacker, making them free-branch instructions. Of course, it is trickier to utilize them since they push an extra return address value onto the stack upon execution. However, it could be possible to discard the extra value in a following gadget, e.g. through a pop operation.

As a final note, the linking section of a gadget has to end with a free branch, but it can also contain additional instructions. For instance, a possible linking section could be the sequence “`popl %ebx; call *%ebx`”, where the first instruction allows the attacker to control the target of the following indirect call.

## 3.2 Gadget Construction

One may argue that, it might not always be possible to find a sufficient number of free-branch instructions in a given binary, or the instructions preceeding a free branch may not have sufficient variety to build a Turing complete set (e.g., a `ret` instruction is almost always preceeded by a function prologue). However, in the x86 architecture, gadgets are not limited to sequences of actual instructions that exist in the program code.

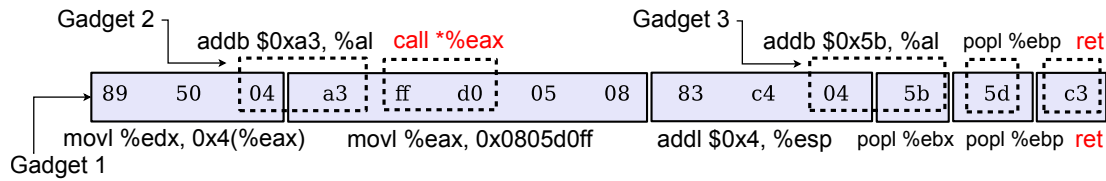


Figure 3.3: Examples of different gadgets that can be extracted from a real byte sequence

Since the IA-32 instruction set does not have fixed-length instructions, the opcode that will be executed depends on the starting point of execution in memory. Therefore, by jumping in the middle of an existing instruction we are very likely to land on another “unintended”, but perfectly legal instruction. Because of the very dense encoding of IA-32 instruction opcodes, this approach yields many different gadgets that an attacker can exploit.

We can use the alignment of the first and the last instruction of the sequence to classify gadgets. In particular, we can identify the following three types of alignments<sup>2</sup>:

- **Aligned gadgets:** These are fully aligned gadgets that only use actual instructions present in the application code. However, they are not limited to the full body of a function but often reuse only the last few instructions before the free branch.
- **Unaligned gadgets:** These gadgets only contain unintended instructions in both the code and the linking section.
- **Aligning gadgets:** These gadgets start by executing unintended instructions but later re-synchronize to the correct instruction alignment in order to reach an aligned free branch.

Figure 3.3 shows how these three different kinds of gadgets can be extracted from a real byte sequence found in `libc`. “Gadget 1” is an aligned gadget that just reuses the

<sup>2</sup>The forth case, i.e. the one that starts with an aligned instruction and finishes with an unintended one, is impossible to achieve because an aligned execution cannot dis-align itself.

entire function tail. “Gadget 2” is an unaligned gadget that contains only unaligned instructions ending with the unintended `call *%eax`. Finally, “Gadget 3” is an aligning gadget, which starts by using an unintended `add` instruction, then re-synchronizes with the normal execution flow, and ends by reaching the function return. This example demonstrates how a short sequence of 14 bytes can be used for constructing many possible gadgets. Considering that a common library such as `libc` contains almost 18K free-branch instructions and that each of them can be used to construct multiple gadgets, it is not difficult for an attacker to find the functionality he needs to execute arbitrary code.

Hence, if we can prevent the attacker from finding useful instruction sequences that terminate with a free branch, we can prevent any return-oriented programming technique. In the next Chapter, we present our approach to reach this goal.

### 3.3 Proposed Defense Techniques against ROP

Several approaches proposed by the research community aim at ensuring the integrity of saved return addresses and, thus, impeding basic ROP attacks. Frantsen et al. [19] presented a shadow return address stack implemented in hardware for the Atmel AVR microcontroller, which can only be manipulated by `ret` and `call` instructions. ROPdefender [23] uses runtime binary instrumentation to implement a shadow return address stack where saved return addresses are duplicated and later compared with the value in the original stack at function exits.

Other approaches [11, 15] aim to detect ROP-based attacks relying on the observation that return-oriented programs result in the execution of short instruction sequences that end with frequent `ret` instructions. They propose to use dynamic binary instrumentation to count the number of instructions executed between two `ret` opcodes. An alert is raised if there are at least three consecutive sequences of five or fewer instructions ending with a `ret`.

While these techniques offer a certain level of protection against the basic ROP attacks, they cannot address the exploits utilizing free-branch instructions different from `ret`. Moreover, many of them suffer from severe performance and deployment issues since runtime binary instrumentation requires monitoring the executed instructions; and others rely on statistical approaches which frequently need to be tweaked as ROP attacks evolve.

The most similar approach to ours is a compiler-based solution developed concurrently and in parallel to our work by Li et al. [22]. This system eliminates unintended `ret` instructions through code transformations, and instruments all `call` and `ret` instructions to implement return address indirection. Specifically, each `call` instruction is modified to push an index value onto the stack that points to a return address table entry, instead of the return address itself. Then, when a `ret` instruction is executed, the saved index is used for looking up the return address from the table. This prevents an attacker from utilizing stack frames containing the addresses of gadgets since saved return addresses are not fetched from the stack.

Although this system is more efficient compared to the previous defenses, it is tailored specifically for compiling kernel code and does not provide a generic defense framework like we do with our approach. Moreover, the implementation requires manual modifications to all the assembly routines. Finally, these techniques can only address the basic ROP attacks that utilize `ret` instructions.

The solution we present in this thesis study is the first to address all free-branch instructions, and the first that can be applied at compile-time to protect any binary executable from ROP attacks.

## Chapter 4

# Our Approach: Code without Gadgets

Our goal is to provide a proactive solution to build gadget-free executables that cannot be targeted by any possible ROP attack. In particular, we strive to achieve a solution that is:

- *comprehensive*: Our solution must eliminate all possible gadgets by removing the linking mechanisms that are necessary to chain instruction sequences together.
- *transparent*: This process must be fully automated and it must require no intervention from the user, such as manual modifications to source code.
- *safe*: The techniques we employ must preserve the semantics of the program, be compatible with compiler optimizations, and support programs that contain routines written in assembly language.

In order to reach our goals, we devise a compiler-based approach that first protects the aligned free-branch instructions to prevent them from being misused by an attacker, and then eliminates all unaligned free-branch instructions inside a binary executable.

We achieve the first point by introducing a mechanism that protects potentially dangerous free-branch instructions by ensuring that they can be executed only if the functions in which they reside were started from their proper entry points. This leaves the attacker with the only option of exploiting unaligned gadgets. We then diminish this possibility through a set of code transformation techniques that ensure free-branch instructions never appear inside any legitimate aligned instruction.

Consequently, an attacker can only execute entire functions from the start to the end as opposed to running arbitrary code. This, effectively, de-generalizes the threat to a traditional return-to-lib(c) attack, eliminating the advantages of achieving Turing completeness without injecting any code in the target process.

Our approach uses a combination of different techniques, namely alignment sleds, return address protection, frame cookies and code rewriting. The rest of this chapter describes each technique in detail.

## 4.1 Free Branch Protection

The first set of techniques aim to protect the aligned free-branch instructions present in the binary. These include the actual `ret` instructions at the end of each function and the `jmp*/call*` instructions that are sometimes present in the code.

Unfortunately, these instructions cannot be easily eliminated without altering the program's behavior. In addition, replacing them with semantically equivalent pieces of code is likely not going to solve the problem because the attacker could still use the replacements to achieve the same functionality.

Therefore, we propose a solution inspired by existing stack protection mechanisms (e.g., StackGuard [13]). The goal is to instrument functions with short blocks of code to make sure that aligned free-branch instructions can only be executed if the running function has been entered from its proper entry point. In particular, we employ two complementary techniques: an efficient return address encryption to protect the `ret`

instructions, and a more sophisticated cookie-based technique we additionally apply only to those functions that contain `jmp*/call*` instructions. These techniques do not require any modifications outside the active stack frame of the function. Thus, our solution can be applied to reentrant or recursive functions, and can be utilized by multi-threaded programs without concern.

Finally, we prepend the code performing the checks with *alignment sleds*. Alignment sleds are special sequences of bytes by which we enforce aligned execution of a set of critical instructions. In particular, we use this technique to prevent an attacker from bypassing our free branch protection code by executing it in an unaligned fashion.

#### 4.1.1 Alignment Sleds

An alignment sled is a sufficiently-long sequence of bytes, encoding one or more instructions that have no effect on the status of the execution. Its length is set to ensure that regardless of the alignment prior to reaching the sled, the execution will eventually land on the sled and execute it until the end. Even if an attacker jumps into the binary at an arbitrary point and executes a number of unaligned instructions, when she reaches the sled, the execution will be forced to realign with the actual code. Thus, it will never reach any unintended opcode present in the instructions following the sled.

The simplest way to implement an alignment sled is to use a sequence of `nop` instructions (see Figure 4.1 for an example). The number of `nop` instructions must be determined by taking into consideration the maximum number of consecutive `nop` bytes (`0x90`) that can tail a valid instruction. If we set the length to anything less than that, an attacker could find an unintended instruction that encompasses the whole sled and any number of bytes from the following instruction, in which case the execution will continue in an unaligned fashion. In the IA-32 instruction set, the longest such sequence becomes possible when we have both an address displacement and an immediate value entirely composed of `0x90` bytes [6], which makes a total of 8 bytes. Additionally, we can have either a ModR/M byte, a SIB byte or an opcode with the

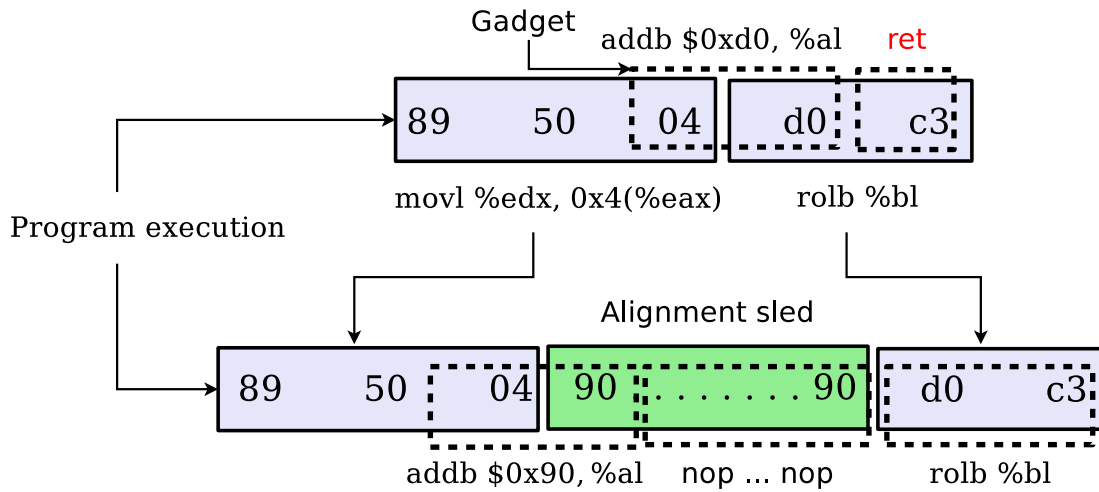


Figure 4.1: Application of an alignment sled to prevent executing an unaligned `ret` (`0xc3`) instruction

value `0x90` (but only one of them at a time). As a result, we can safely set the number of `nop` instructions in our sled to 9.

Note that the sled length calculation presented in this section is an over-approximation. By also taking into account the bytes preceding the sled and which instructions they can possibly encode, it is possible to automatically compute the required sled length case-by-case.

We also prepend the sled with a relative jump instruction to skip over the sled bytes. Consequently, if the execution is already aligned, it will hit the jump and not incur the performance penalty of executing the sequence of `nop` instructions.

### 4.1.2 Return Address Protection

Return address protection involves instrumenting the entry points of functions that contain `ret` instructions with a short header that encrypts the saved return address stored on the stack. Before each return instruction, we then insert a corresponding footer to restore the saved return address to its original value. If an attacker jumps into a function at an arbitrary position and eventually reaches our footer, the decryption

routine processes the unencrypted return address provided by the attacker, computes an invalid value and the following return instruction attempts to transfer the execution flow to an incorrect address that the attacker cannot control. The encryption method we utilize is a simple exclusive-or of the return address with a *random key* generated at runtime. Since this solution does not affect the layout of the stack in any way, it does not require any further modifications to the function code.

### 4.1.3 Frame Cookies

In order to prevent the attacker from using existing `jmp*/call*` instructions, we need to adopt a similar strategy as we did with `ret` instructions. On the one hand, we aim to permit correct execution of `jmp*/call*` only if the function execution starts from a valid entry point; but on the other hand we cannot directly make use of the previous return address encryption scheme. That is because, in the previous case, encryption provided an implicit means to regulate the target of the control flow transition, and we did not need to perform any check on the computed return value – a wrong value automatically resulted in an invalid memory access. However, with `jmp*/call*` the target destination is given in a register or a memory location, and thus we must explicitly control it after doing the necessary checks to ensure proper execution of the function.

To this end, we instrument the entry points of the functions that contain `jmp*/call*` instructions with an additional header to compute and push a random cookie onto the stack. This cookie is an exclusive-or of a *random key* generated at runtime and a *per-function constant* generated at compile-time. The constant is used for uniquely identifying the function and it does not need to be kept secret.

Next, we prepend all the `jmp*/call*` instructions with a *validation block* which fetches the cookie, decrypts it, and compares the result with the *per-function constant*. If the cookie is not found or the values do not match, we invalidate the jump/call destination causing the application to crash. Finally, in the function footer, we insert a

simple instruction to remove the cookie from the stack.

A significant consequence of using this technique is that we end up altering the layout of the stack by storing an additional value inside. This requires us to fix the memory offsets of some of the instructions that access the stack according to the location where we store the cookie. We discuss the details of this issue in Chapter 5.

## 4.2 Code Rewriting

The second set of techniques we adopt in our approach focus on removing any unaligned free-branch instructions.

In the IA-32 instruction set, instructions consist of some or all of the following fields: instruction prefixes, an opcode, a ModR/M byte, a SIB (Scale-Index-Base) byte, an address displacement, and finally, an immediate value. A `ret` instruction can be encoded with any of the `0xc2`, `0xc3`, `0xca` or `0xcb` bytes, and as such, can be part of any of the instruction fields (excluding the prefixes). On the other hand, `jmp*/call*` instructions are encoded by two-byte opcodes: the `0xff` opcode followed by an ModR/M byte carrying certain three-bit sequences. Hence, in addition to appearing inside a single instruction, they can also be obtained by a combination of two bytes coming from two consecutive instructions.

In this section, we discuss the various cases and describe the code rewriting techniques we use to eliminate all unintended free-branch opcodes.

### 4.2.1 Register Reallocation

The ModR/M and the SIB bytes are used for encoding the addressing mode and the operands of an instruction. The use of certain registers as operands cause either the ModR/M or the SIB byte to be set to a value that corresponds to a `ret` opcode. The

| ModR/M | Operand 1      | Operand 2      |
|--------|----------------|----------------|
| 0xc2   | %eax, %ax, %al | %edx, %dx, %dl |
| 0xc3   | %eax, %ax, %al | %ebx, %bx, %bl |
| 0xca   | %ecx, %cx, %cl | %edx, %dx, %dl |
| 0xcb   | %ecx, %cx, %cl | %ebx, %bx, %bl |

Table 4.1: ModR/M values encoding `ret` opcodes

| SIB  | Base | Scaled Index |
|------|------|--------------|
| 0xc2 | %edx | %eax*8       |
| 0xc3 | %ebx | %eax*8       |
| 0xca | %edx | %ecx*8       |
| 0xcb | %ebx | %ecx*8       |

Table 4.2: SIB values encoding `ret` opcodes

possible undesired encodings of these bytes are shown in Table 4.1 and Table 4.2. For instance, an instruction that specifies `%eax` as the source operand and `%ebx` as the destination, such as “`movl %eax, %ebx`”, assigns the value `0xc3` to the ModR/M byte. Similarly, using `%edx` as the base and `(%ecx * 8)` as the scaled index, the instruction “`addl $0x2a, (%edx, %ecx, 8)`” would contain `0xca` in its SIB byte.

In order to eliminate the unintended `ret` opcodes that result from such circumstances, we must avoid all of the undesired register pairings listed in Table 4.1 and Table 4.2. We achieve this by manipulating the register allocation performed during compilation to ensure that those pairs of registers never appear together in a generated instruction. When we detect such an instruction, we can perform the compiler’s register allocation stage again, this time enforcing a different register assignment. As an alternative, we can perform a local reallocation by temporarily swapping the contents of the original operand with a new register, and then rewriting the instruction with this new register as its operand. In this way, we can bring forth an acceptable register pairing for the same instruction.

In some cases, the ModR/M byte could be used to specify an opcode extension and a single register operand. In such cases certain pairs of opcode extensions and register operands could encode a `ret` byte in a similar fashion (e.g., “`addl $0xFFFFFFFF, %ebx`” is encoded as “`0x81 0xc3 0xff 0xff 0xff 0xff`”). It is possible to rewrite these instructions using the same techniques described above to replace the register operand with a different one and eliminate the `ret` opcode.

Similarly, most floating point instructions are encoded by an opcode and an opcode extension given inside the ModR/M byte. FPU instructions that have certain opcode extensions and that operate on either of the floating point registers `%st(2)` or `%st(3)` cause the `ret` byte `0xc2` or `0xc3` to appear in their ModR/M byte respectively (e.g., `fld %st(2)` is represented as `0xd9 0xc2`). Since many FPU instructions use implicit operands that cannot be substituted with different ones, applying register reallocation techniques in these cases is not trivial. However, all such instructions can have the `ret` opcode only in their second bytes. Therefore, we prepend them with an alignment sled leaving only the instruction’s opcode between the sled and the unintended `ret`. This makes it impossible for the attacker to create any gadget utilizing this unintended `ret`.

### 4.2.2 Instruction Transformations

`ret` bytes appear in the opcodes encoding the instructions `movnti` (`0x0f 0xc3`) and `bswap` (`0x0f 0xc8+<register_identifier>`). In the first case, the instruction `movnti` acts like a regular `mov` operation except that it uses a *non-temporal hint* to reduce cache pollution. Thus, we can safely replace it with a regular `mov` without any significant consequence. For the second, the opcode is determined according to the operand register and can encode a `ret` byte when certain registers are specified as the operand; consequently, as described in the previous section, we can perform a register reallocation to choose a different operand and obtain a safe `bswap` opcode.

### 4.2.3 Jump Offset Adjustments

Jump and call instructions may contain free-branch opcodes when using immediate values to specify their destinations. For instance, `jmp .+0xc8` is encoded as “`0xe9 0xc3 0x00 0x00 0x00`”.

A free-branch instruction opcode can appear at any of the four bytes constituting the jump/call target. If the opcode is the least significant byte, it is sufficient to append the forward jump/call with a single `nop` instruction (or prepend it if it is a backwards jump/call) in order to adjust the relative distance between the instruction and its destination<sup>1</sup>:

$$\text{jmp } .+0xc8 \quad \Rightarrow \quad \begin{array}{l} \text{jmp } .+0xc9 \\ \text{nop} \end{array}$$

However, when the opcode is at a different byte position, the number of `nop` instructions we need to insert increase drastically (256 for the second, 64K for the third and 16M for the last byte).

Although free-branch opcodes commonly appear at the second byte of the jump/-call destinations (and inserting 256 NOPs is still a feasible solution), for the higher order bytes, the situation is highly uncommon. For example, a jump offset encoded by “`0x00 0x00 0xc3 0x00`” would indicate a 12MB forward jump. Considering the fact that jump instructions are ordinarily used for local control flow transitions inside a function, a 12MB offset would be infeasible in practice. Even if we were to come across such an offset, possibly in call targets, we can relocate the functions or code chunks addressed by the instruction to remove the opcodes.

---

<sup>1</sup>In the x86 architecture, all immediate jump targets are specified relative to the following instruction. Only indirect jumps use absolute targets.

### 4.2.4 Immediate and Displacement Reconstructions

Several arithmetic, logic and comparison operations can take immediate values as an operand, which may contain free-branch instruction opcodes. We can remove these by substituting the instruction with a sequence of different instructions that construct the immediate value in steps while carrying the same semantics. The following examples demonstrate the reconstruction process, assuming that `%ebx` is free or has been saved beforehand:

```

addl $0xc2, %eax  ⇒  addl $0xc1, %eax
                    inc %eax

                    movb $0xc9, %bl
xorb $0xca, %al   ⇒  incb %bl
                    xorb %bl, %al

```

Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as “0x88 0x45 0xca”). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence that uses an adjusted displacement value to avoid the undesired bytes. We achieve this by setting the displacement value to a safe value and then compensating for our changes by temporarily adjusting the value in the base register. For example, we can perform a reconstruction such as:

```

                    incl %ebp
movb $0xa1, -0x36(%ebp) ⇒ movb %al, -0x37(%ebp)
                    decl %ebp

```

### 4.2.5 Inter-Instruction Barriers

Unintended `jmp*/call*` opcodes can result from the combination of two consecutive instructions. This happens when the last byte of an instruction is `0xff` and the

first byte of the following instruction encodes a suitable opcode extension. We can remove these unintended `jmp*/call*` opcodes by inserting a *barrier* between two such instructions, effectively separating them and destroying the unintended opcode. For the barrier, the trivial choice of a `nop` instruction is not suitable since an `0xff` followed by a `0x90` still encodes an indirect call. Thus, we have to choose a safe nop-like alternative, such as `movl %eax, %eax`.

### 4.3 Limitations of the Approach

By applying the techniques presented in this section, we can remove all unaligned free-branch instructions from the application binary, and protect the aligned ones from being misused by an attacker. However, since our protection mechanism does not remove the aligned free branches, but instead prepends a short piece of code to protect them, the result of the compilation will still contain some gadgets.

In fact, an attacker may attempt to skip the alignment sled by directly jumping *into* the return address or indirect jump/call protection blocks. This may result in executing a useful instruction sequence (intended or unintended) which terminates at the free-branch instruction we protect without actually executing the checks in our protection block.

However, since our approach only requires inserting two very short pieces of code, the number of possible gadgets that can be built is very limited and the gadget sizes are restricted to few instructions. By keeping this issue in mind, it is therefore possible to specifically craft the return address and indirect jump/call protection blocks to make sure they do not contain any convenient gadgets.

We discuss the particular techniques we used in our prototype implementation, and the number and type of gadgets that are left in the binaries compiled by our tool in Chapter 5.

# Chapter 5

## Implementation

In this section, we describe G-Free, a prototype system we developed based on the techniques presented in Chapter 4, and we discuss some of the issues we encountered while compiling `glibc` using our prototype.

Our implementation efforts primarily focus on creating a fully-automated system that would not require any modifications to the program’s source code or to the existing compilation tools. Unfortunately, system-wide libraries, which are the primary targets of ROP attacks, often rely on hand-tuned assembly routines to perform low-level tasks. This makes a pure compiler-based solution unable to intercept part of the final code. Therefore, we implemented our prototype in two separate components: an assembly code pre-processor designed to work as a wrapper for the GNU Assembler (`gas`), and a simple binary analyzer responsible for gathering some mandatory information that is not available in the assembly source code.

## 5.1 Assembly Code Pre-Processor & Binary Analyzer

The assembly code pre-processor intercepts the assembly code generated by `cc1` (the GNU C compiler included in the GNU Compiler Collection) or coming directly from an assembly language source file. It then performs the required modifications to remove all the possible gadgets, and finally passes the control to the actual `gas` assembler.

We must stress that in this implementation we modify neither the compiler nor the assembler; both are completely oblivious to the existence of our pre-processing stage. We only replace the `gas` executable with a small wrapper that is responsible for invoking our pre-processor before executing the assembler.

Our system successfully handles assembly routines written using non-standard programming practices. Moreover, it supports compiler optimizations, including all of the GCC standard optimization levels (in fact, `glibc` *does not* compile if GCC optimizations are disabled).

There is one significant implication of directly working with assembly code: Our pre-processor is not exposed to the numeric values of immediate operands and memory displacements since these are often represented by symbolic values until linkage. Thus, it is not possible for us to identify all of the instructions that contain unintended free-branch opcodes just by looking at the assembly code. In order to address this issue, we use a two-step compilation approach. First, our system compiles a given program without doing any modifications to the original code. During this compilation, our pre-processor tags each of the instructions that contain immediate values or displacements with unique symbols. This information is then exported in the final executable's symbol table. In a second step, we use a binary analyzer to read the symbol table of the executable and check whether any of the instructions pointed to by our tagged symbols needs to be rewritten because it contains unaligned free-branch instructions. This analysis produces a log of the tags corresponding to the instructions

we need to modify. This log is consumed afterwards by the pre-processor during a second compilation phase in order to provide it with the previously missing information. The entire process is performed in an automated manner.

## 5.2 Obtaining Random Keys

As described in Chapter 4, our approach requires a random value to encrypt both the return address and the cookie stored on the stack. For this purpose, our prototype inserts a key generation routine at the beginning of the program's entry point (or initialization routine if it is a library). In our prototype, this routine simply reads a 32-bit random value from the Linux special file */dev/random* and stores the value in a global memory location.

If the attacker has a way to read arbitrary memory locations before performing the actual attack, he could be able to fetch the per-process random key and use it to craft the required values on the stack to defeat our implementation. This limitation is common to many canary-based stack protection mechanisms such as StackGuard [13] and ProPolice [20]. However, this problem can be avoided by substituting the per-process random key with a per-function key computed at runtime in the function headers.

## 5.3 Stack Reference Adjustments

We store our cookie just above the saved return address in the stack, shifting the frame base upwards by 4 bytes. Since a function usually uses the `%ebp` register to reference the stack relative to the frame base, and our cookie is located below the frame base, references to the stack local variables remain unchanged. On the contrary, references to function parameters which are stored below the frame base, and therefore below our cookie, need to be adjusted by 4 bytes (See Figure 5.1).

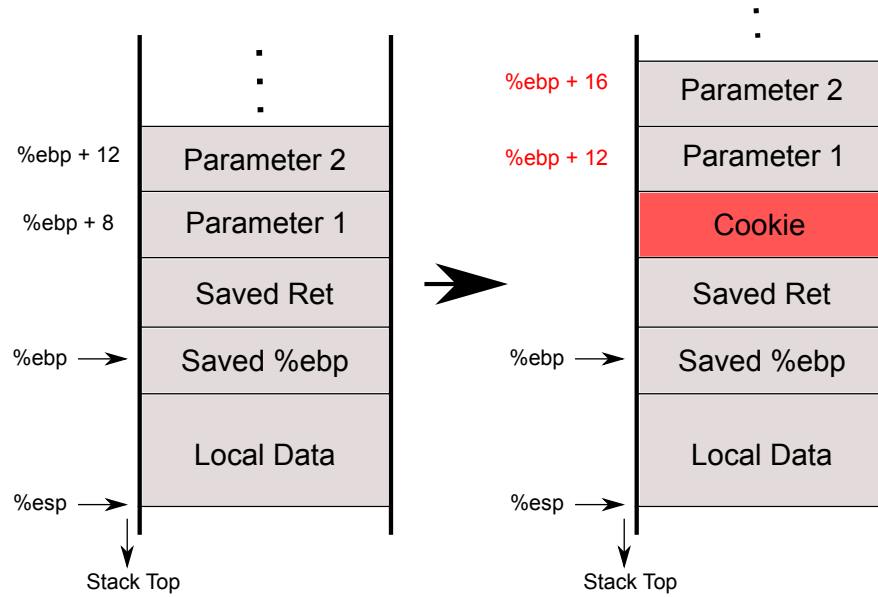


Figure 5.1: Stack frame before and after inserting a cookie

We achieve this by simply correcting each positive displacement to  $\%ebp$  by adding to it the size of our cookie:

```
movl 0x8(%ebp), %eax  ⇒  movl 0xc(%ebp), %eax
```

Note that compiler optimizations that adopt Frame Pointer Omission (FPO) use the stack pointer to reference arguments and local variables. In this case, we need to compute the displacement of the stack pointer to the function's frame at any given position in the function in order to identify and fix the references and locate our cookie in the stack. This requires a comprehensive stack depth analysis. We have designed our pre-processor to perform this analysis on-the-fly without the need for any extra pass over the source file, even when the execution flow of the processed function is non-linear. We keep track of `push` & `pop` operations and arithmetic computations on the stack pointer and update the system's view of stack depth accordingly. Additionally, we record the current stack depth at each jump instruction together with the jump destination, and then restore the state when we arrive at the recorded destination, which successfully simulates analyzing different execution branches separately for changes to

the stack pointer. Depending on the state of the stack, we can then determine whether a stack access (e.g., `120(%esp)`) points to a local variable or to a function's parameter, so that we can apply the displacement adjustment where appropriate.

## 5.4 Conditional Code Rewriting

Our prototype implements all immediate and displacement reconstruction strategies we described in Chapter 4. However, to reduce the performance overhead, we apply those transformations only when absolutely necessary. Otherwise, we use a faster approximate solution. In particular, during the compilation, we prepend each instruction that contains free-branch opcodes among its immediate or displacement fields with an alignment sled in order make it impossible to reach the unintended opcodes inside. This allows us to avoid the performance costs of multiple-step immediate/displacement reconstruction instructions and get away with executing a single-cycle `jmp` instruction.

However, this approach might not always be sufficient to remove all possible gadgets; merely the bytes of the protected instruction could still encode a number of valid instructions ending with a free-branch opcode and an attacker could jump *into* the protected instruction to utilize these as a gadget (e.g., `"pushl $0xffc38913" - 68 89 13 c3 ff` includes the gadget `"movl %edx, (%ebx); ret" - 89 13 c3`). In order to address such issues, our system automatically checks these bytes after the compilation. If it detects that they do indeed contain valid instructions, it falls back to the safer (but slightly less efficient) immediate or displacement reconstruction methods. This process does not require a recompilation; we use the space occupied by the sled to write the new gadget-free instruction sequence.

|    |             |       |                  |
|----|-------------|-------|------------------|
| 50 |             | pushl | %eax             |
| a1 | 00 f0 fd b7 | movl  | 0xb7fdf000, %eax |
| 31 | 44 24 04    | xorl  | %eax, 0x4(%esp)  |
| 58 |             | popl  | %eax             |

Figure 5.2: Code inserted to protect the aligned return instructions

|    |             |       |                    |
|----|-------------|-------|--------------------|
| 50 |             | pushl | %eax               |
| a1 | 00 f0 fd b7 | movl  | 0xb7fdf000, %eax   |
| 35 | 76 1f 0f 0f | xorl  | \$0x0f0f1f76, %eax |
| 39 | 45 04       | cmpl  | %eax, 0x4(%ebp)    |
| 58 |             | popl  | %eax               |
| 74 | 02          | jz    | freebranch         |
| 31 | d2          | xorl  | %edx, %edx         |

freebranch:

|    |    |     |       |
|----|----|-----|-------|
| ff | e2 | jmp | *%edx |
|----|----|-----|-------|

Figure 5.3: Code inserted to protect the aligned indirect jump/call instructions

## 5.5 Return Address and Indirect Jump/Call Protection Blocks

As previously explained in Chapter 4, our solution protects aligned free-branch instructions by introducing two short blocks of code: the return address protection block and the indirect jump/call protection block (the current implementations are shown in Figure 5.2 and Figure 5.3). These two pieces of code are the only ones in the final executable that can still contain gadgets, and therefore, they must be carefully designed to prevent any possible attack.

The return address protection code is 11 bytes long and all bytes are under our control, with the exception of the 4-byte address of the random key, which could change for each compiled program and for shared libraries at each relocation. To ensure that

the code is safe to use, we need to prevent this value from containing potentially dangerous instructions. In our implementation, we control the least significant two bytes by automatically inserting appropriate alignment directives into the assembled code when defining the key storage location, ensuring that the address always ends with the innocuous “0xf0 0x00” sequence. In addition, according to the Linux process memory layout, the most significant address byte of the `.bss` section (where we store our random key) is limited in practice to 0x08 for regular ELF executables and 0xb\* for shared libraries<sup>1</sup>. Therefore, it encodes either a variation of a *load immediate into register* instruction (e.g., “`mov $IMM, %reg`”), or an `or` instruction between two 8-bit operands. All in all, only the third byte can still be considered arbitrary. We do not further attempt to control that value; because even in the worst possible scenario, it would introduce just a single fixed gadget in the whole executable. This obviously cannot satisfy the requirements of Turing completeness, such as the ability to modify memory locations and to implement conditional jumps.

The indirect jump/call protection block is 19 bytes long and contains an additional 4-byte dynamic section: the per-function constant identifier we generate at compile time to compute the cookie. The example shown in Figure 5.3 (that uses a 0x0f0f1f76 function identifier) is entirely gadget-free because it contains no aligned or unaligned instruction sequences that would make it possible for an attacker to reach `jmp *%edx` without invalidating its contents. In fact, any logic/arithmetic operation that does not yield a result of zero (e.g., `incl %ebp`, unless `%ebp` overflows) clears the zero flag in the processor and would prevent the use of the conditional jump `jz .+4`<sup>2</sup>. Consequently, the value inside `%edx` is cleared.

Different values of the function identifier could potentially introduce a new and useful gadget; but since these constants can be arbitrarily chosen and do not need to be kept secret, we can easily work around problematic cases. In order to minimize the risk in the first place, we use simple heuristics such as using bytes that represent invalid

<sup>1</sup>The Linux process memory layout dictates that dynamic shared libraries are loaded at the address range 0xc0000000–0x40000000, starting from higher addresses. As a result, in practice almost any shared library has 0xb\* as the most significant address byte of its `.bss` section.

<sup>2</sup>The conditional jump `jz` only jumps when the zero flag is set in the processor

| Gadget 1 |    |      |             |
|----------|----|------|-------------|
| 00       | f0 | addb | %dh, %al    |
|          | fd | std  |             |
| b7       | 31 | movb | \$0x31, %bh |
| 44       |    | incl | %esp        |
| 24       | 04 | andb | \$0x04, %al |
| 58       |    | popl | %eax        |

| Gadget 2 |    |          |             |
|----------|----|----------|-------------|
| f0       | fd | lock std |             |
| b7       | 31 | movb     | \$0x31, %bh |
| 44       |    | incl     | %esp        |
| 24       | 04 | andb     | \$0x04, %al |
| 58       |    | popl     | %eax        |

| Gadget 3 |    |      |             |
|----------|----|------|-------------|
| 04       | 58 | addb | \$0x58, %al |

Figure 5.4: Gadgets available in the return address protection block

opcodes (e.g., `0x0f 0x0f`) and avoiding dangerous opcodes such as those encoding `mov` or free-branch instructions.

Figure 5.4 and Figure 5.5 shows all the gadgets that can be extracted from our current system implementation. As can be seen, apart from the ability to load the `%eax` with a controlled value (`popl %eax`), the gadgets have no value.

## 5.6 Compiling `glibc`

During our case study of compiling `glibc` using G-Free, we have encountered several issues requiring particular care. These were mostly related to unconventional programming practices used for dealing with low-level tasks, or manually optimized assembly

|             |    |      |             |
|-------------|----|------|-------------|
| 45          |    | incl | %ebp        |
| 04          | 58 | addb | \$0x58, %al |
| 74          | 02 | jz   | freebranch  |
| 31          | d2 | xorl | %edx, %edx  |
| freebranch: |    |      |             |
| ff          | e2 | jmp  | *%edx       |

Figure 5.5: Gadget available in the indirect jump/call protection block

code. This section explains our observations in this regard, and explains how we cope with these special cases.

### 5.6.1 Multiple Entry Points

We have come across various functions in `glibc` that include more than one possible entry point. Our system successfully detects such functions and instruments all entry points with the appropriate headers. Additionally, we prepend each header that lies in the execution path of other entry points with a jump instruction to skip over the header, ensuring that only one header is executed per function call.

### 5.6.2 Functions that Access the Saved Return Address

In `glibc`, we have encountered a single function, namely `setjmp` that accesses the saved return address on the stack. `setjmp`, together with the function `longjmp`, is used for implementing *non-local jumps*: a call to `setjmp` saves the current stack context to restore it afterwards when `longjmp` is invoked. This behavior conflicts with our return address protection scheme. Since the return address is stored in an encrypted form on the stack, a call to `setjmp` saves the encrypted return address and a subsequent call to `longjmp` results in an illegal memory access. In order to solve this problem, we modified our prototype to detect when the saved return address

is moved to a register and perform the decryption on the duplicated value to ensure correct functionality.

### 5.6.3 Jumps between Functions

In numerous cases, a function directly jumps to another one without saving the return address, essentially making that jump an exit point. During compilation, we check every jump destination to recognize jumps outside the current function and treat them as regular exit points for inserting the necessary footers. These footers are not meant to protect a free-branch instruction, since none follows, but to restore the return address to its original value before transferring the execution flow to another function.

### 5.6.4 Position Independent Code (PIC) Addressing Issues

Shared libraries, including `glibc`, need to be compiled as position independent executables (PIE) so that they work correctly regardless of the address they are loaded at in the memory. Such executables cannot directly access global data; because their address is not known until the code is loaded.

To overcome this obstacle, a data structure, called the Global Offset Table (GOT), is responsible for storing the absolute addresses of global data. Programs then access GOT and use its entries to locate the desired pieces of data. However, GOT cannot be directly referenced for the same reasons as well.

Thus, PIEs compute the absolute address of the GOT programmatically by first getting the current value of the instruction pointer, and then adding to it the relative distance to GOT. The following example gives an excerpt from `glibc` source code demonstrating this process:

```
call __i686.get_pc_thunk.cx
addl $_GLOBAL_OFFSET_TABLE_, %ecx
```

The function `__i686.get_pc_thunk.cx` stores the address of the following instruction in `%ecx`. Then the symbol which will resolve to the relative distance between the `addl` instruction and the GOT is added to `%ebx`, to get the absolute address of the GOT.

Remember that, if an immediate value contains a free-branch opcode we prepend it with an alignment sled. Considering the given example, if `$_GLOBAL_OFFSET_TABLE_` contains such an opcode and we insert a sled in between the instructions, we end up breaking the absolute value computation. This is because, when we reach the `addl` instruction, `%ecx` does not contain the current address anymore, it falls short by the length of the sled. In order to fix this problem, we detect all such computations and correct the `add` operation to compensate for the length of the sled. For instance, the above `addl` instruction becomes:

```
addl $_GLOBAL_OFFSET_TABLE_+<sled_length>, %ecx
```

## 5.7 Limitations of the Implementation

When inserting a `nop` at a certain position to fix a jump offset, we may actually affect the offsets of many other jumps since this process alters the whole address space of the binary. Our prototype binary analyzer does not consider the overall structure of the binary file when reporting the instructions to fix. Therefore, while fixing a set of jump offsets, several other offsets may start to contain free-branch opcodes. This makes it necessary to perform several compilations until all the offsets are fixed.

Note that in this process, we may need to fix a single jump instruction several times. However, since inserting `nop` instructions between a jump and its destination can only increase the offset but never decrease it, we are sure to find a safe offset after a finite number of iterations.

A more optimized analyzer that can perform a global analysis and take into account

the target of every jump instruction would eliminate this problem. It would also produce smaller executables since the various recompilations may insert many otherwise unnecessary `nop` instructions.

# Chapter 6

## Evaluation

The main goal of our evaluation is to show that our solution can be applied for compiling real-world applications and producing gadget-free executables. To demonstrate that we are able accomplish this goal, we compiled `glibc` and a number of common Linux applications using G-Free. We then investigated the impact of our binary modifications on the file size of compiled files and performed a set of experiments to measure the incurred performance overhead.

In our tests, we used G-Free in conjunction with `gas 2.20` and `GCC 4.4.3`. All the experiments were performed on a 2GHz Intel Core 2 Duo T7300 machine with 2GB of memory, running Arch Linux (i686) with Linux kernel 2.6.33.

### 6.1 Compilation Results

Since ROP attacks usually extract their gadgets from common libraries, we focus our evaluation on `glibc` version 2.11.1. The original library compiled without G-Free contains 9921 `ret` instructions (6106 of which unaligned) and 8018 `jmp*/call*` instructions (6602 of which unaligned). This sums up to almost 18K free-branch op-codes, each of which can be potentially used by an attacker to build many different

| Program Name | Version       | Number of unaligned |            | Number of functions |          |
|--------------|---------------|---------------------|------------|---------------------|----------|
|              |               | ret                 | jmp*/call* | with RAP            | with JCP |
| glibc        | 2.11.1        | 6106                | 6602       | 2817                | 827      |
| gzip         | 1.4           | 433                 | 410        | 122                 | 10       |
| grep         | 2.6.3         | 523                 | 369        | 174                 | 20       |
| dd           | coreutils-8.5 | 252                 | 181        | 95                  | 8        |
| md5sum       | coreutils-8.5 | 203                 | 86         | 68                  | 3        |
| ssh-keygen   | openssh-5.5p1 | 607                 | 712        | 271                 | 20       |
| lame         | 3.98.3        | 2228                | 1342       | 669                 | 28       |

Table 6.1: Statistics on binaries compiled with G-Free (RAP=Return Address Protection, JCP=indirect Jump/Call Protection)

gadgets.

After we compiled `glibc` using our system, all unintended `ret` and `jmp*/call*` instructions were either removed or made ineffective by prepending them with an alignment sled. In addition, all aligned free-branch instructions were protected by adding our return address and indirect jump/call protection blocks. As a result, the library compiled with G-Free contained only the four types of gadgets we have previously presented in Figure 5.4 and Figure 5.5.

However, due to the newly inserted code and instruction rewriting techniques, the size of the gadget-free version of the library increased by 30%. Although this value might appear to be high, one third of the overhead is caused by `nop` instructions included in the alignment sleds. As already discussed in Chapter 5, most of these could be eliminated by a more optimized implementation.

Unfortunately, providing a gadget-free version of `glibc` is not sufficient to completely prevent ROP attacks, since the attacker could still build the gadget set from other libraries or the application binary itself. Therefore, to achieve a complete protection against ROP, it is necessary to compile the entire application and all its libraries with our technique. To demonstrate that our tool can be applied to this more general scenario, we include in our evaluation a number of common Linux applications.

| Program Name | Size (KBs)       |                | Overhead |
|--------------|------------------|----------------|----------|
|              | Original Version | G-Free Version |          |
| glibc        | 1320.4           | 1728.4         | 30.9%    |
| gzip         | 72.7             | 92.4           | 27.0%    |
| grep         | 86.3             | 106.3          | 23.2%    |
| dd           | 48.0             | 57.9           | 20.6%    |
| md5sum       | 30.9             | 37.7           | 22.1%    |
| ssh-keygen   | 140.6            | 182.5          | 29.7%    |
| lame         | 322.6            | 406.6          | 26.0%    |

Table 6.2: The increase in size for binaries compiled with G-Free

Table 6.1 shows the number of unaligned free-branch instructions and the number of functions requiring protection blocks that were available in the original binaries of the programs we have performed our tests on. The last two columns of Table 6.1 show that most of the functions can be protected by our efficient return address encryption technique while very few of them required the more complex indirect jump/call protection block. This is a consequence of the fact that, according to what we observed in our experiments, programs rarely use `jmp*/call*` instructions.

Table 6.2 lists the increase in size for each program compiled with G-Free compared to the size of the original program binary. Our solution was able to provide gadget-free binaries with an average size increase of 25.9% (more than half of which were caused by redundant `nop` instructions).

## 6.2 Performance Measurements

Table 6.3 shows the performance overheads we measured by running the different applications compiled with our prototype (this includes the gadget-free versions of the programs and all their linked libraries). For each application, we designed program-specific test cases, which are summarized as:

| Program Name | Execution Time (seconds) |                | Overhead |
|--------------|--------------------------|----------------|----------|
|              | Original Version         | G-Free Version |          |
| gzip         | 66.5                     | 68.4           | 2.9%     |
| grep         | 81.3                     | 82.9           | 2.0%     |
| dd           | 86.6                     | 88.9           | 2.6%     |
| md5sum       | 82.5                     | 82.9           | 0.6%     |
| ssh-keygen   | 51.2                     | 53.6           | 4.6%     |
| lame         | 115.5                    | 122.0          | 5.6%     |

Table 6.3: Performance comparisons of common Linux applications when the application and all the linked libraries are compiled with G-Free

- *gzip*: Compressing a 2GB file with random content.
- *grep*: Performing regular-expression-based string searches in a 2GB file.
- *dd*: Reading from the Linux special file “/dev/zero” and writing to a regular file 2G of zero bytes.
- *md5sum*: Computing the md5 hash of a 2GB file.
- *ssh-keygen*: Generating 100 2048-bit RSA keys.
- *lame*: Encoding a 10-minute wav file into mp3 format.

The average performance overhead was 3.1% – a value comparable with the overhead caused by well known stack protection systems such as StackShield [38] and StackGuard [13].

Since a library cannot be run as a standalone program, we evaluated the performance overhead of our gadget-free version of `glibc` using a set of well-known benchmarks. In particular, we used the Phoronix Test Suite [3] which provides one of the most comprehensive benchmark sets for the Linux platform. Table 6.4 lists a sample of the benchmarks that represent different application categories such as games, mathematical and physical simulations, 3D rendering, disk and file system activities,

compression, and well-known server applications. The results indicate that the performance overhead of an application using our gadget-free version of `glibc` is on average 1.09%.

| Benchmark                  | Original Libc | G-Free Libc | Overhead |
|----------------------------|---------------|-------------|----------|
| FS-Mark (Files/s)          | 15.1          | 14.9        | 1.3%     |
| IOzone-read (MB/s)         | 23.0          | 22.7        | 1.4%     |
| IOzone-write (MB/s)        | 22.8          | 22.6        | 0.4%     |
| Minion (s)                 | 250.2         | 250.7       | 0.2%     |
| TSCP (Nodes/s)             | 224642.0      | 224385.0    | 0.1%     |
| BYTE (Lines/s)             | 7288371.3     | 6948792.8   | 4.6%     |
| PHP Comp (s)               | 102.9         | 107.3       | 4.3%     |
| Unpack Linux Kernel (s)    | 30.30         | 31.01       | 2.3%     |
| BZIP2 (s)                  | 65.63         | 65.84       | 0.3%     |
| Ogg Encoding (s)           | 27.14         | 27.20       | 0.2%     |
| dcraw (s)                  | 52.68         | 52.99       | 0.6%     |
| Timed MAFFT (s)            | 52.48         | 52.55       | 0.1%     |
| SQLite (s)                 | 189.09        | 191.78      | 1.4%     |
| x2642009 (Frames/s)        | 13.72         | 13.62       | 0.7%     |
| x11perf (Operations/s)     | 912000        | 912000      | 0.0%     |
| OpenArena (Frames/s)       | 46.93         | 46.67       | 0.6%     |
| FFmpeg (s)                 | 24.93         | 25.02       | 0.4%     |
| OpenSSL (Signs/s)          | 25.28         | 25.28       | 0.0%     |
| John The Ripper (Real C/S) | 1854667       | 1857333     | 0.1%     |
| Continued on next page     |               |             |          |

| Benchmark                 | Original Libc | G-Free Libc | Overhead |
|---------------------------|---------------|-------------|----------|
| Timed HMMer Search (s)    | 88.93         | 89.31       | 0.4%     |
| Bwfirt (s)                | 284.9         | 285.3       | 0.2%     |
| Dbench (MB/s)             | 83.7          | 82.0        | 2.0%     |
| Sudokut (s)               | 97.1          | 100.4       | 3.5%     |
| GMPbench (Score)          | 2955.5        | 2954.5      | 0.03%    |
| PyBench (s)               | 6791.0        | 6959.0      | 2.5%     |
| 7-Zip (MIPS)              | 2822.0        | 2802        | 0.7%     |
| LZMA (s)                  | 291.67        | 291.86      | 0.01%    |
| FLAC Audio Encoding (s)   | 12.96         | 13.09       | 1.0%     |
| Himeno (MFLOPS)           | 152           | 151.44      | 0.4%     |
| Bullet Physics Engine (s) | 39.58         | 39.74       | 0.4%     |
| PostgreSQL (Trans/s)      | 155.24        | 156.66      | 0.9%     |
| Apache(Requests/s)        | 7129.05       | 6836.24     | 4.1%     |
| GtkPerf (s)               | 20.89         | 20.49       | 1.9%     |
| Urban Terror (Frames/s)   | 34.20         | 34.05       | 0.9%     |
| C-Ray (s)                 | 553.7         | 554.0       | 0.05%    |
| GraphicsMagick (Iter/min) | 45            | 44          | 2.2%     |
| Gcrypt Library (micros)   | 6963          | 6983        | 0.3%     |
| GnuPG (s)                 | 20.46         | 20.67       | 1.0%     |
| Average:                  | 1.09%         |             |          |
| Std:                      | 1.27          |             |          |

Table 6.4: Performance comparison of the original and G-Free `glibc` using benchmarks from the Phoronix Test Suite

## Chapter 7

### Conclusions

Return-oriented programming is an evolution of the return-to-lib(c) attack technique and it recently attracted significant attention from the scientific community. Even though much research has been conducted on the topic, no comprehensive defense has been proposed to date.

With this thesis study, we propose a novel, comprehensive solution to defend against return-oriented programming by removing all gadgets from a program binary at compile-time. Our approach targets all possible free-branch instructions, and therefore, is independent from the techniques used for linking the gadgets together.

We implemented our solution in a prototype called G-Free, a pre-processor for the GNU Assembler that does not require any modifications to any of the compilation tools. Our experiments show that G-Free is able to remove all gadgets at the cost of a very low performance overhead and an acceptable increase in the file size.

G-Free provides a general defense framework targeting both regular applications and system libraries, it is fully automated, and is compatible with compiler optimizations. Hence, we satisfy our design goals of building a system that is *comprehensive*, *transparent* and *safe*.

# Bibliography

- [1] <http://marc.info/?l=openbsd-misc&m=105056000801065>.
- [2] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. <http://support.microsoft.com/kb/875352/EN-US/>.
- [3] Phoronix Test Suite. <http://www.phoronix-test-suite.com/>.
- [4] ROP attack against Data Execution Prevention technology, 2009. <http://www.h-online.com/security/news/item/Exploit-s-new-technology-trick-dodges-memory-protection-959253.html>.
- [5] Symantec: Internet Security Threat Report. [http://www4.symantec.com/Vrt/wl?tu\\_id=jLac123913792490340803](http://www4.symantec.com/Vrt/wl?tu_id=jLac123913792490340803), 2009.
- [6] Intel 64 and IA-32 Architectures Software Developer's Manuals. <http://www.intel.com/products/processor/manuals/>, 2010.
- [7] Aleph One. Smashing The Stack for Fun and Profit. In *Phrack Magazine n.49*, 1996.
- [8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings*

- of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.
- [9] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. In *Proceedings of EVT/WOTE 2009. USENIX/ACCURATE/IAVoSS*, 2009.
- [10] S. Checkoway and H. Shacham. Escape from Return-Oriented Programming: Return-Oriented Programming without Returns (on the x86). Technical report, 2010.
- [11] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting Return-Oriented Programming Malicious Code. In *Lecture Notes in Computer Science*, 2009.
- [12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Usenix Security Symposium*,, 2003.
- [13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7<sup>th</sup> USENIX Security Symposium, USA*, 1998.
- [14] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition*,, 2000.
- [15] L. Davi, A. R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *Proceedings ACM workshop on Scalable trusted computing*, 2009.
- [16] Felix Lidner. CONFidence 2.0. Developments in Cisco IOS forensics.
- [17] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *Proceedings of CCS*, 2008.

- [18] A. Francillon, D. Perito, and C. Castelluccia. Defending Embedded Systems Against Control Flow Attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, 2008.
- [19] M. Frantsen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of USENIX security*, 2001.
- [20] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-Smashing Attacks (ProPolice). In <http://www.trl.ibm.com/projects/security/ssp/>, 2003.
- [21] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium, USA*, 2009.
- [22] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with Return-less Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, 2010.
- [23] M. W. Lucas Davi, Ahmad-Reza Sadeghi. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. Technical report, Technical Report HGI-TR-2010-001.
- [24] M. Conover, w00w00 Security Team. w00w00 on Heap Overflows. Technical report, <http://www.w00w00.org/files/articles/heaptut.txt>, 1999.
- [25] Nergal. The advanced return-into-lib(c) exploits. In *Phrack Magazine n.58*, 2001.
- [26] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

- [27] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25<sup>th</sup> Annual Computer Security Applications Conference (ACSAC)*, Honolulu, Hawaii, USA., pages 60–69. IEEE Computer Society, Dec. 2009.
- [28] Scut, Team Teso. Exploiting Format String Vulnerabilities. 2001.
- [29] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique, 2005. <http://www.suse.de/~krahmer/no-nx.pdf>.
- [30] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSES. In *Proceedings of Operating System Symposium SOSP*, 2007.
- [31] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.
- [32] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CSS)*, 2004.
- [33] Solar Designer. return-to-libc attack. Technical report, bugtraq, 1997.
- [34] The PaX Team. PaX Address Space Layout Randomization. Technical report, <http://pax.grsecurity.net/docs/aslr.txt>.
- [35] The PaX Team. PaX non-executable pages. Technical report, <http://pax.grsecurity.net/docs/noexec.txt>.
- [36] Tim Kornau. Return Oriented Programming for the ARM Architecture. Technical report, Master’s thesis, Ruhr-Universität Bochum, 2010.
- [37] A. van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. Technical report, [http://people.redhat.com/mingo/exec-shield/docs/WHP0006US\\_Execshield.pdf](http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf).

- [38] Vendicator. StackShield: A “Stack Smashing” Technique Protection Tool for Linux. Technical report, <http://www.angelfire.com/sk/stackshield/>.
- [39] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS*, 2009.