On Generating Gadget Chains for Return-Oriented Programming

Vom Fachbereich Informatik (FB20) an der
Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation von
Andreas Follner, M.Sc. aus Wien

1. Referent: Prof. Eric Bodden, PhD
2. Referent: Asst. Prof. Dr. Mathias Payer
3. Referent: Prof. Dr. Mira Mezini

Tag der Einreichung: 7. November 2016
Tag der Disputation: 21. Dezember 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Center for Research in Security and Privacy
Secure Software Engineering Group

Darmstadt 2016
Hochschulkennziffer: D17

On Generating Gadget Chains for Return-Oriented Programming

Genehmigte Dissertation von Andreas Follner aus Wien.

1. Referent: Prof. Eric Bodden, PhD
2. Referent: Asst. Prof. Dr. Mathias Payer
3. Referent: Prof. Dr. Mira Mezini

Tag der Einreichung: 7. November 2016
Tag der Disputation: 21. Dezember 2016

Darmstadt 2016 - Hochschulkennziffer: D17

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 7. November 2016.

Andreas Follner

Abstract

With the increased popularity of embedded devices, low-level programming languages like C and C++ are currently experiencing a strong renewed interest. However, these languages are *unsafe*, meaning that programming errors may lead to *undefined behaviour*, which, in turn, may be exploited to compromise a system's integrity. Many programs written in these languages contain such programming errors, most infamous of which are buffer overflows. In order to fight this, there exists a large range of mitigation techniques designed to hinder exploitation, some of which are integral parts of most major operating systems' security concept. Even the most sophisticated mitigations, however, can often be bypassed by modern exploits, which are based on the principle of code reuse: they assemble, or *chain*, together existing code fragments (known as *gadgets*) in a way to achieve malicious behaviour. This technique is currently the cornerstone of modern exploits.

In this dissertation, we present ROPocop, an approach to mitigate code-reuse attacks. ROPocop is a configurable, heuristic-based detector that monitors program execution and raises an alarm if it detects suspicious behaviour. It monitors the frequency of indirect branches and the length of basic blocks, two characteristics in which code-reuse attacks differ greatly from normal program behaviour. However, like all mitigations, ROPocop has its weaknesses and we show that it and other similar approaches can be bypassed in an automatic way by an aware attacker. To this end, we present PSHAPE, a practical, cross-platform framework to support the construction of code-reuse exploits. It offers two distinguishing features, namely it creates concise semantic summaries for gadgets, which allow exploit developers to assess the utility of a gadget much quicker than by going through the individual assembly instructions. And secondly, PSHAPE automatically composes gadgets to construct a chain of gadgets that can invoke any arbitrary function with user-supplied parameters. Invoking a function is indeed the most common goal of concurrent exploits, as calling a function such as `mprotect` greatly simplifies later steps of exploitation.

For a mitigation to be viable, it must detect actual attacks reliably while at the same time avoiding false positives and ensuring that protected applications remain usable, i.e., do not crash or become very slow. In the tested sample set of applications, ROPocop detects and stops all twelve real attacks with no false positives. When executed with ROPocop,

real-world programs exhibit only some slight input lag at startup but otherwise remain responsive. Yet, we further show how PSHAPE can be used to fully automatically create exploits that bypass various mitigations, for example, ROPocop itself. We also show gadgets PSHAPE found easily, that have great relevance in real exploits, and which previously required intense manual searches to find. Lastly, using PSHAPE, we also discovered a new and very useful gadget type that greatly simplifies gadget chaining.

Zusammenfassung

Mit der Verbreitung eingebetteter Systeme erleben hardwarenahe Programmiersprachen wie C und C++ einen großen Aufschwung. Allerdings sind diese Sprachen *unsicher*, was bedeutet, dass Programmierfehler zu sogenanntem *undefiniertem Verhalten* führen können. Dies kann ausgenutzt werden, um ein System zu kompromittieren. Viele Programme, die in C/C++ geschrieben sind, beinhalten solche Programmierfehler, zu welchen beispielsweise Pufferüberläufe zählen. Um diese Gefahr zu bekämpfen, existieren diverse Abwehrmechanismen, von denen einige in unseren Betriebssystemen integriert sind und einen wichtigen Beitrag zur Systemsicherheit leisten. Allerdings können selbst die ausgeklügeltsten Abwehrmechanismen umgangen werden, wie aktuelle Angriffe zeigen. Diese Angriffe basieren auf dem Konzept, vorhandene Programmfragmente neu zusammenzusetzen, um bösartigen Code zu erzeugen. Diese Technik ist der Grundstein moderner Angriffe.

In dieser Dissertation präsentieren wir ROPocop, eine Methode um solche Angriffe zu verhindern. ROPocop ist ein auf einer Heuristik basierendes, konfigurierbares Programm, das den Programmfluss eines anderen Programms zur Laufzeit analysiert und Alarm auslöst, falls es ungewöhnliches Verhalten feststellt. Es analysiert Eigenschaften, in denen sich reguläres von bösartigem Programmverhalten, bei dem vorhandene Programmfragmente neu zusammengesetzt werden, unterscheidet. Wie alle aktuellen Abwehrmechanismen hat auch ROPocop Schwächen und wir zeigen, wie ROPocop und ähnliche Abwehrmechanismen vollautomatisiert umgangen werden können. Hierfür präsentieren wir PSHAPE, ein plattformübergreifendes Framework, welches die Entwicklung von Angriffen unterstützt. PSHAPE hilft in zweierlei Hinsicht: es erzeugt kompakte, semantische Zusammenfassungen für Programmfragmente. Diese erlauben eine schnelle Feststellung der Auswirkungen, die ein Fragment auf den Programmzustand hat. Außerdem ist PSHAPE in der Lage, Programmfragmente vollautomatisch zusammenzufügen, um einen Exploit zu erzeugen, der eine Funktion mit beliebigen Parametern aufruft. Dieses Verhalten ist realitätsnah, da das Aufrufen einer Funktion wie `mprotect` nachfolgende Schritte der Exploit-Entwicklung stark vereinfacht.

Ein praxistauglicher Abwehrmechanismus muss Angriffe zuverlässig erkennen und löst im Idealfall keinen falschen Alarm aus. Außerdem muss sichergestellt sein, dass das ge-

schützte Programm bedienbar bleibt, das heißt, nicht stark verlangsamt wird oder gar abstürzt. Wir haben ROPocop mit zwölf realen Exploits getestet, die alle zuverlässig erkannt wurden, ohne dass ROPocop einen Fehlalarm auslöste. Programme laufen, geschützt durch ROPocop, zu Beginn leicht verlangsamt, danach jedoch ohne merkbare Verzögerungen. Des Weiteren zeigen wir, wie PSHAPE vollautomatisch Exploits erzeugt, die in der Lage sind, diverse Abwehrmechanismen, wie beispielsweise ROPocop, zu umgehen. Außerdem zeigen wir, dass es bestimmte Programmfragmente, die bisher unter großem Zeitaufwand manuell gesucht werden mussten, zuverlässig und vollautomatisch findet. Zudem findet PSHAPE eine neue Art von Programmfragment, das die Exploit-Entwicklung stark vereinfacht.

Acknowledgments

First of all, I would like to thank my advisor Eric Bodden for his his tremendous support over the years. Even during the most busy times he always made time to guide me in the right direction. I also want to thank Mathias Payer for his much appreciated feedback on this dissertation and various papers, as well as our fruitful discussions. Similarly, much gratitude goes to Alexandre Bartel for his advice as well as his significant contributions to several papers.

I am also very grateful to my defense committee: Eric Bodden, Mathias Payer, Mira Mezini, Marc Fischlin, Matthias Hollick, and Oskar von Stryk.

Special thanks go to all my colleagues for the interesting discussions and for contributing to a great work environment. Special, special thanks go to Kevin Falzon, whom I shared the office with and whose taste in music made it easy to find good tracks to play when we both worked late. And of course, a big thank you goes to Andrea Püchner and Karina Köhres, who made sure things run smoothly at the office.

Furthermore, I would like to thank my parents Peter and Sylvia, my brother Christian, my grandparents Gertrude, Maria, and Franz, and my aunt Ursula for always believing in me as well as their support and words of encouragement. The same goes for all my friends around the world, especially Stefan and Arseni.

Last but definitely not least I want to thank my girlfriend Mandy for her unlimited love, support, and patience.

Contents

1 Introduction **1**

1.1 Research Motivation . . . 3

1.2 Solution Overview . . . 4

1.3 Contributions . . . 5

1.4 Publications . . . 6

1.5 Outline . . . 6

2 Background **7**

2.1 Basics of the x86-64 Architecture . . . 7

2.1.1 Registers . . . 7

2.1.2 Memory Management . . . 10

2.1.3 Function Calls . . . 12

2.2 Bugs and Vulnerabilities . . . 16

2.2.1 Buffer Overflow . . . 16

2.2.2 Use-After-Free . . . 20

2.2.3 Type Confusion . . . 27

2.3 Exploit Mitigation . . . 30

2.3.1 Executable Space Protection . . . 30

2.3.2 Address Space Layout Randomization . . . 31

2.3.3 Stack Canaries . . . 33

2.3.4 Control-Flow Integrity . . . 33

2.3.5 Variable Reordering . . . 34

2.4 Code-Reuse Attacks . . . 35

2.4.1 Return-Oriented Programming . . . 37

2.4.2 Jump-Oriented Programming . . . 39

2.4.3 Gadgets . . . 40

2.5 Summary . . . 43

3 Related Work **45**
3.1 Mitigations . . . 45
3.2 Attacks . . . 55
3.3 Gadgets and ROP Chains . . . 60

4 Gadget Chaining **63**
4.1 ROP in Practice . . . 64
4.2 Environment Restrictions . . . 67
4.2.1 AntiCRA . . . 67
4.2.2 Impact on Exploit Development . . . 69
4.2.3 DEP+ . . . 71
4.2.4 Evaluation . . . 73
4.2.5 Summary . . . 79
4.3 Automated Gadget Chaining . . . 79
4.3.1 PSHAPE . . . 80
4.3.2 Gadget Summaries . . . 80
4.3.3 Gadget Chaining . . . 82
4.3.4 Implementation . . . 84
4.3.5 Comparison with Existing Tools . . . 85
4.3.6 PSHAPE in Practice . . . 89
4.3.7 Summary . . . 92
4.4 Gadget Selection . . . 92
4.4.1 Assessing Gadget Quality . . . 93
4.4.2 Discussion of the Metrics . . . 100
4.4.3 Evaluation . . . 101
4.4.4 Summary . . . 105
4.5 Summary . . . 105

5 Cross System Case Studies **107**
5.1 Case Study 1 . . . 108
5.2 Case Study 2 . . . 109
5.3 Case Study 3 . . . 113
5.4 Performance . . . 114
5.4.1 Gadget Summaries . . . 114
5.4.2 Gadget Chaining . . . 117
5.4.3 Time Savings due to GaLity . . . 117
5.5 Discussion . . . 118

6 Future Directions **121**
6.1 Short-term 121
6.2 Long-term 122

7 Conclusion **125**
7.1 Summary and contributions 125

List of Figures

2.1 Change of the stack pointer by `push` and `pop` instructions.[1] 11
2.2 Stack with one stack frame 15
2.3 Stack with two stack frames 15
2.4 Stack before and after a buffer overflow. 21
2.5 Buffer overflow 21
2.6 Exploited buffer overflow with shellcode before the return address 22
2.7 Exploited buffer overflow with shellcode after the return address 22
2.8 Object with virtual functions 25
2.9 Use after free exploitation timeline 26
2.10 vtable corruption attack 26
2.11 vtable injection attack 27
2.12 vtable reuse attack 28
2.13 Type confusion memory layout 29
2.14 Type confusion vulnerability 30
2.15 Different memory layouts between reboots due to ASLR 32
2.16 Stack canary 34
2.17 Different stack layout due to variable reordering 36
2.18 ROP 37
2.19 ROP with data intertwined 38
2.20 JOP overview 40

4.1 Structure of real ROP exploits 65
4.2 A simple ROP chain to `VirtualProtect` 66
4.3 Analysis of the number of indirect branches in a row and the lowest average basic block length of our test set 74
4.4 Performance of ROPocop 79
4.5 Gadget summary 82
4.6 Overview of PSHAPE 83
4.7 Gadget Chain for apache2 90

4.8 Gadget Chain for nginx 91
4.9 Gadget Chain for chrome.exe 91
4.10 Gadget increase due to MPX 103
4.11 Gadget distribution with and without MPX 104

5.1 Gadget chain with long gadgets for Apache 2. 109
5.2 Gadget chain with long gadgets for nginx. 110
5.3 Universal heuristic breaker in `ntdll.dll`. 113
5.4 Call-preceded gadgets 114
5.5 Gadget Chain for nginx setting up six parameters using only call-preceded gadgets. 115
5.6 gdb output of gadget shown in Figure 5.5 116
5.7 Gadget Chain for Apache 2 setting up six parameters using only call-preceded gadgets. 116

6.1 Summaries of a `cmovne rax, rbx ; ret` gadget. 123

List of Tables

2.1 x86-64 registers and subregisters 8
2.2 Instruction sizes and subregisters 8

3.1 Overview of CFI implementations 48
3.2 Overview of heuristic-based mitigations 49
3.3 Overview of vtable protection schemes 51
3.4 Overview of ASLR improvements 53
3.5 Overview of ASLR attacks 57

4.1 Estimated number of gadgets in various environments 65
4.2 Analysis of Exploits and Programs 77
4.3 Summary of ROP tools 86
4.4 Gadget Extraction 89
4.5 Gadget Chains 89
4.6 Gadget Categories 94
4.7 Rules for grading instructions 98
4.8 Evaluation of GaLity 102

Listings

2.1 Conditional assembler instruction 10
2.2 C-equivalent of Listing 2.1 10
2.3 `push` equivalent 11
2.4 `pop` equivalent 11
2.5 Function prologue and epilogue 13
2.6 Function call 14
2.7 Variable declaration 17
2.8 Space allocation 18
2.9 Out of bounds access in an array 18
2.10 A buffer overflow vulnerability 18
2.11 Fixed buffer overflow 20
2.12 Use-after-free bug #501572 in openssl 23
2.13 Virtual functions 24
2.14 Virtual function dispatch 25
2.15 Unsafe downcast 29
2.16 C code to show variable reordering 35
2.17 JOP dispatcher gadget 40
2.18 JOP dispatcher gadget 40
2.19 Disassembly of `488D450F490F43C2482BD8` 41
2.20 Disassembly of `8D450F490F43C2482BD8` 41
2.21 Disassembly of `450F490F43C2482BD8` 42
2.22 Disassembly of `0F490F43C2482BD8` 42
2.23 Disassembly of `490F43C2482BD8` 42
2.24 Disassembly of `0F43C2482BD8` 42
2.25 Disassembly of `43C2482BD8` 42
5.1 Bad heuristic breaker 112
5.2 Good heuristic breaker 112

Chapter 1

Introduction

Low-level memory-corruption vulnerabilities date back to 1988 when the Morris worm [155] spread across the internet. While there are no official numbers, it is widely believed that it infected 10% of computers connected to the internet, at that time an estimated 6,000 computers. Since then, other infamous worms spawned, such as Code Red (2001) [79], Slammer (2003) [46], or Blaster (2003) [54], which exploited similar vulnerabilities. While this lead to increased awareness of such bugs, they are still common in today's software and modern exploits still abuse the same types of vulnerabilities that existed almost 30 years ago.

The underlying problems are twofold: due to the Von-Neumann architecture [166], code and data are not separated in memory, which means that a program can be tricked into executing data. The second problem concerns programming languages: unsafe languages like C and C++ are still widely used, due to their high performance and because there is a large amount of legacy code. The problem with those languages is that they allow the programmer to directly access and manipulate memory, as well as putting the responsibility to sanitize user input into the hand of the programmer. For example, if the programmer declares a buffer of 20 characters, the programmer has to check that the user's input is not longer than 20 characters. If the programmer neglects this, longer input overwrites other data in memory, which can lead to wrong program output, crashes, or, in the worst case, allow an attacker to execute arbitrary code.

Up until the early 2000s, when mitigation techniques were not widely deployed, exploitation was simple and consisted of the following steps: (i) inject shellcode[1] into the target program. Since almost all programs take some form of user input, this was usually not a problem. (ii) leverage a buffer overflow vulnerability to overwrite a code pointer, such as a return address. This attack, widely known as *stack smashing* [3], allows hijacking the intended control-flow of the program because the attacker can force the program to con-

[1]Traditionally, shellcode is code that opens a shell, but nowadays, shellcode is used as an umbrella term describing any code the attacker wants to run.

tinue execution at an arbitrary location. Normally, the attacker would overwrite the code pointer with the memory address of the code injected in the previous step, which will then be executed.

To battle these exploits, large numbers of mitigation techniques, designed to prevent exploitation of such vulnerabilities, have been proposed by academia and industry alike. A few of them are now standard on modern architectures, such as data execution prevention (DEP) [4], which prevents the execution of injected code, and address-space-layout-randomization (ASLR) [80], which makes it hard for the attacker to know, where her shellcode is in memory. While they are often effective, attackers still bypass them regularly, developing more and more sophisticated attack techniques. Indeed, it is often the case that a new mitigation is bypassed just weeks after its adoption.

One attack technique, called return-oriented programming (ROP) [135,144], has become a cornerstone of today's exploits, and is based on the principle of code-reuse. DEP prevents the execution of data, making it impossible for an attacker to inject and run arbitrary code. However, the exploitation of the underlying vulnerability, hence also the control-flow hijacking, is not prevented, allowing an attacker to jump to any code that already exists in memory. Due to the specifics of certain architectures, including the popular x86-64 architecture, which we focus on in this dissertation, it is possible to chain carefully selected code-fragments or *gadgets* together, which then execute consecutively. This combination of ROP gadgets is called a ROP chain.

Because ROP is Turing complete [135, 144], it is in theory possible to use ROP to construct any arbitrary program. This is, however, complex, cumbersome, and requires a lot of manual labour and attention to detail. Therefore, exploit developers normally avoid this path. Instead, they use ROP to invoke an API that allows them to execute injected shellcode, effectively bypassing DEP. Such exploits are often referred to as two-staged: the first stage employs ROP to bypass DEP, using an operating system API such as `VirtualProtect` (Windows) or `mprotect` (Linux). These APIs exist because the system needs a way to change memory protection levels at runtime, e.g., when a program dynamically generates code. After this first stage, the second stage, which is regular shellcode, executes. In practice, we are only aware of one pure ROP attack [95], as these attacks are much more difficult to create and provide no benefit over two-staged exploits.

Such two-staged attacks are simpler to construct than a pure ROP attack because they only need to use ROP to invoke one function. Yet, their construction nevertheless requires a large amount of manual effort. Currently available tools designed to support ROP exploit development often contain bugs, have limited functionality, are outdated, or do not support modern architectures. This limits their usefulness in practice, and the fact that those tools are only used for the simple task of gadget discovery is testament to that.

On the other hand, many mitigations proposed by academia drastically reduce the number of available gadgets in a binary, forcing exploit developers to use more complex gadgets. Complex gadgets are often long and typically contain side-effects, i.e., do not only achieve the task originally desired by the exploit developer, but also affect memory or registers in an undesired way. Furthermore, complex gadgets often have preconditions, for example, require a certain register to point to readable memory in order for the gadget to work correctly. Currently available tools perform badly in current scenarios, and cannot work at all when such mitigations are used, making ROP exploit development a predominantly manual task, with very basic tool-support.

In this dissertation, we look at both the mitigation and construction of ROP exploits. First, we present an approach to detect ROP exploits, called ROPocop. Next, we propose an attempt to automate ROP exploit development, even in the face of state-of-the-art mitigation techniques, called PSHAPE.

1.1 Research Motivation

ROP is an essential technique used in today's exploits, which makes it a large threat to a system's integrity. A large body of work has been proposed to mitigate this problem, however, none of the approaches are widely adopted and new attacks have already shown that many of them can be bypassed. Clearly, there is a need for further investigation on how ROP can be hindered effectively.

At the same time, more sophisticated mitigations make ROP exploit development increasingly difficult. However, currently, tool support for this task is very limited, especially in regards to bypassing mitigations. This topic has not been researched very well so far, despite it being a topic interesting to a large audience. Indeed, ROP exploit development is a task frequently attempted not only by blackhat hackers for malicious purposes, but also by security professionals, penetration testers, researchers, or security analysts. For example, researchers might aim to implement a secure compiler that generates code which contains a minimal amount of useful gadgets. To evaluate whether this helps in practice, they may attempt to build a ROP exploit using the remaining gadgets. The process of developing ROP exploits is currently a manual, time-consuming task and the outcome is directly related to the skill and experience of the analyst. A tool, on the other hand, is deterministic, hence delivering reproducible and comparable results. Tool support, however, is very limited and has several issues. Many tools currently available are outdated and do not support the omnipresent x86-64 architecture. Other tools are not maintained any longer, contain bugs or simply do not work as expected. Lastly, many tools are restricted to gadget discovery. While an essential task, various other features are desirable. For example, gadget discovery

usually produces a large text file often containing millions of gadgets. From this file, the exploit developer has to manually pick the gadgets she needs and assemble them, i.e., put the gadgets in the correct order to achieve her goal.

Automating this process would provide a large benefit to exploit developers. It saves time and allows them to focus on other steps of exploit development, such as bypassing other mitigation techniques. However, automation introduces challenges: one main issue regarding automation is that of state explosion. Millions of gadgets and no assumptions about the exploit developer's goal make almost all random combinations of gadgets useful in some way or the other, which results in billions of usable ROP chains. Clearly, creating and analysing all these chains is infeasible.

Furthermore, all current tools output gadgets in assembler code. While showing every single instruction in a gadget provides a lot of detail, this high level of detail is often not required. Instead, gadgets could be summarized and translated into a more human-readable form. Determining the effect of a gadget that consists of 20 or more instructions can take even an experienced analyst some time but, even worse, is error prone. Considering that binaries usually contain millions of gadgets, this is a time-consuming process.

Hence, we identify two main problems in current ROP chain development: state explosion, caused by millions of gadgets and no assumptions about the exploit developer's goals, and the lack of a more compact and easy to read representation of gadgets. If done right, such a representation allows the developer to search for specific effects a gadget has.

To summarize, both the mitigation of ROP exploits, as well as the construction of ROP exploits, leave room for improvement.

1.2 Solution Overview

We address the problems stated above as follows.

To mitigate ROP exploits, we propose ROPocop, a configurable, heuristic-based detector that continuously monitors program execution. ROPocop monitors the frequency and length of basic blocks, which are characteristics in which ROP attacks, and code-reuse attacks in general, often differ greatly from regular program behaviour. The detection thresholds can be configured freely, to fine-tune them to the application that requires protection. To increase protection even further, we enforce non-executable data regions similar to DEP, however, without a way to bypass it. More precisely, we only allow the instruction pointer to point inside a binary, but never to other mapped memory.

To assist in the construction of ROP exploits, we propose PSHAPE. PSHAPE addresses the challenges raised before as follows.

State explosion. As we explained in the previous chapter, attacks are almost exclusively two-staged, i.e., they use ROP to invoke a function that allows the execution of injected code. This approach requires the exploit developer to correctly initialize arguments to the appropriate API. Usually, the exploit developer provides the arguments in memory and then uses gadgets to move the parameters from memory into the correct registers used for passing parameters to functions. Therefore, by making the realistic assumption that an exploit developer wants to invoke a function, one can discard gadgets that do not move data between memory or registers. Since this still leaves many gadgets to consider, we propose a further reduction of the search space by analysing the remaining gadgets, assessing their usefulness and quality, and then keeping only the most suitable gadgets. These gadgets are then permuted and the resulting gadget chains analysed.

Compact gadget representation. To achieve a more compact and human-readable form, we propose gadget summaries, consisting of a gadget's preconditions and postconditions. Preconditions are requirements that have to be satisfied for the gadget to run correctly, for example, that a register has to point to readable memory. Postconditions describe the state of registers and memory after a gadget has executed.

We implemented this approach in a tool called PSHAPE. PSHAPE extracts gadgets and runs a *deep gadget analysis*, where gadgets are converted to an intermediate representation (IR), semantically analysed, and summarized. This summary is a compact representation of a gadget's pre- and postconditions. Based on this summary, gadgets not useful for loading data into registers are discarded. Then, PSHAPE runs a *gadget quality assessment* algorithm called GaLity to determine the quality of each gadget. Lastly, PSHAPE uses *smart gadget permutations* to chain the most suitable gadgets together with the goal of ensuring a specified number of registers is loaded with data the exploit developer can control. PSHAPE also works well against restrictions imposed by mitigations, such as ROPocop and various others. Since we show how various complex mitigations, including ROPocop, can be bypassed completely automatically with PSHAPE, our thesis statement concentrates on the automation part of this dissertation:

Coalescing deep gadget analysis, smart gadget permutation, and gadget-quality assessment enables feasible automation of gadget chaining even in the presence of mitigations.

1.3 Contributions

To summarize, this dissertation makes the following original contributions.

- ROPocop: A prototype for an exploit mitigation system. It consists of DEP+, software-enforced data-execution prevention which cannot be bypassed using operating systems APIs, and AntiCRA, a configurable, heuristic-based code-reuse attack detector.

- PSHAPE: A framework for automated ROP chain generation. It analyses gadgets, creates summaries, and chains gadgets together fully automatically.
- GaLity: A set of metrics for assessing gadget quality. It allows PSHAPE to reduce the search space to make gadget chaining feasible.

1.4 Publications

Parts of this work have been published at conferences, journals, and workshops. Our mitigation technique ROPocop [64] (Chapters 4.2.1, 4.2.2, 4.2.3, and 4.2.4), was published in the Elsevier Journal of Information Security and Applications (JISA). PSHAPE [63], responsible for creating gadget summaries and chaining gadgets together (Chapters 4.3.2, 4.3.3, 4.3.4 , 4.3.5, and 4.3.6), was published at the International Workshop on Security and Trust Management (STM), co-located with ESORICS. The metrics to grade gadgets (Chapters 4.4.1, 4.4.2, and 4.4.3) were implemented in a tool called GaLity [62] and published at the International Symposium on Engineering Secure Software and Systems (ESSoS), where it also received an Artifact Evaluation Award[2]. Lastly, a talk on exploit automation, covering all three publications, was given at DeepSec[3].

1.5 Outline

The remainder of this dissertation is organized as follows. Chapter 2 covers background knowledge required to be able to understand the remainder of this work. It covers certain aspects of the x86-64 architecture, such as registers, memory management, and function calls, three classes of vulnerabilities which are prime targets for exploitation, exploit mitigation, and code-reuse attacks. Chapter 3 discusses related work. Chapter 4.1 explains how code-reuse attacks are used in practice and elaborates upon restrictions that make code-reuse attacks more difficult. In Chapter 4.2 we present ROPocop, which represents such a restriction. Next, in Chapter 4.3 we discuss how the process of gadget chain creation can be automated, presenting PSHAPE. Lastly, in Chapter 4.4, we introduce GaLity, the set of metrics that assesses the quality of gadgets, which is necessary so the search space for automation can be limited in a sensible way. Chapter 5 presents three case studies, involving all three of our tools. Chapter 6 discusses ideas for future directions, and Chapter 7 concludes.

[2] `http://www.artifact-eval.org/`
[3] `https://deepsec.net/`

Chapter 2

Background

In this chapter we introduce relevant background information which also serves as a means for the reader to look up concepts referred to later in this dissertation. Chapter 2.1 discusses basics of the x86-64 architecture [84, 100]. It does not, by any means, claim completeness and instead focuses on topics relevant for this dissertation: registers, function calls and calling conventions, and memory management (i.e., heap and stack). Chapter 2.2 introduces buffer overflow, use-after-free, and type confusion vulnerabilities and how they are exploited. Chapter 2.3 covers mitigation techniques, concentrating on those implemented in popular operating systems (OS). Lastly, Chapter 2.4 covers code-reuse attacks, a sophisticated technique, which is the cornerstone of today's exploits.

2.1 Basics of the x86-64 Architecture

In this chapter we introduce basics of the x86-64 architecture, also referred to as AMD64, IA-32e, or EM64T. It covers topics such as registers, management of stack and heap, and function calls.

2.1.1 Registers

Registers are extremely fast memory storages (orders of magnitude faster than RAM and Cache) used to hold data for operations or memory pointers. x86-64 has 16 *general purpose registers* which are 64 bit long, and shown in Table 2.1. Addressing only parts of a whole 64-bit general purpose register is possible, and often done when it is not necessary to use the whole register. If, for example, the value `42h` should be stored in `rax`, it makes no difference whether this value is stored in `rax`, `eax`, `ax`, or `al`, because it occupies only seven bit (`42h` = `1000010b`). The size of the generated instruction, however, is greatly affected, as Table 2.2 shows. Therefore, compilers will usually choose the smallest possible subregister when optimizations are enabled.

Table 2.1: x86-64 registers and subregisters

64-bit register (bits 63…0)	Lower 32 bits (Bits 31…0)	Lower 16 bits (bits 15…0)	Higher 8 bits (bits 15…8)	Lower 8 bits (bits 7…0)
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si	–	sil
rdi	edi	di	–	dil
rbp	ebp	bp	–	bpl
rsp	esp	sp	–	spl
r8	r8d	r8w	–	r8b
r9	r9d	r9w	–	r9b
r10	r10d	r10w	–	r10b
r11	r11d	r11w	–	r11b
r12	r12d	r12w	–	r12b
r13	r13d	r13w	–	r13b
r14	r14d	r14w	–	r14b
r15	r15d	r15w	–	r15b

Table 2.2: Instruction sizes and subregisters

Instruction	OpCode	Size
mov rax, 42h	48C7C042000000	7
mov eax, 42h	B842000000	5
mov ax, 42h	66B84200	4
mov al, 42h	B042	2

Despite these registers being dubbed "general purpose", two of them have a clear intended usage.

- `rsp`, the stack pointer, always points to the top of the stack, a data structure to store data such as variables, introduced in Chapter 2.1.2.
- `rbp`, the base pointer or frame pointer, also points to the stack (more precisely, it points at the bottom of the current stack frame. We discuss this in more detail Chapter 2.1.3) and is usually used to access local variables and parameters, with local variables being at a negative offset and parameters being at a positive offset of `rbp`. Compilers can, however, decide to not use a base pointer, in which case `rbp` can be used like any other general purpose register and therefore hold arbitrary information. This is an optimization referred to as *base pointer omission*. In that case, variables on the stack are accessed using offsets of `rsp`, which makes manual program analysis more difficult.

Another very important register is `rip`, the *instruction pointer* or *program counter*. It always points to the instruction that is executed next, and is therefore responsible for the program-flow (see Chapter 2.1.3). `rip` cannot be a changed directly, e.g., by using it in conjunction with a `mov` instruction, but only through certain instructions. For example, `ret` loads the value `rsp` points to into `rip` and increases `rsp` by 8 bytes, mimicking a `pop rip` instruction. The most important instructions that can change `rip` are `ret`, `call`, `jmp`, and `loop`.

The last register important to this work is the `rflags` register, which is in essence a collection of flags. Those flags store, among other data, information about results of previous operations. For example, if an operation's result is zero, the Zero flag `ZF` is set (`1`), if it is not zero, `ZF` is cleared (`0`). Some of these flags are used to determine control flow using condition instructions.

Consider the example in Listing 2.1[1]. The `cmp` instruction compares two values by subtracting them, discarding the result, and setting flags in `rflags` accordingly. Remember that `ZF` is set, if the result of an operation is zero. Therefore, in this example, if `rax` and `rbx` contain the same value, `ZF` is set, otherwise cleared. The next instruction, `cmovnz rcx, rbx` writes the value of `rbx` in `rcx` if `ZF` is cleared[2]. Otherwise, the instruction is not executed. Listing 2.2 shows equivalent C code.

Many other registers exist on the x86-64 architecture, such as registers specifically for floating point instructions (`xmm0` through `xmm15`) or debug registers. However, none of

[1]Note that throughout this dissertation we use Intel syntax for assembler code, i.e., the order of parameters for an instruction is destination before source. Parameter size is determined by the name of the register that is used.

[2]The `cmovnz` instruction can be read as ***c**onditional **mov**e if **n**ot **z**ero*.

Listing 2.1: Conditional assembler instruction

```
cmp rax, rbx
cmovnz rcx, rbx
```

Listing 2.2: C-equivalent of Listing 2.1

```
if (rax != rbx) {
        rcx = rbx; }
```

these are relevant to this work, which is why we omit them. The Intel Architecture Software Developer's Manual [84] contains more detailed information. To summarize, in the context of this dissertation, the important registers are the 16 general purpose registers, with `rsp` and `rbp` having a special meaning, the instruction pointer `rip`, and the `rflags` register.

2.1.2 Memory Management

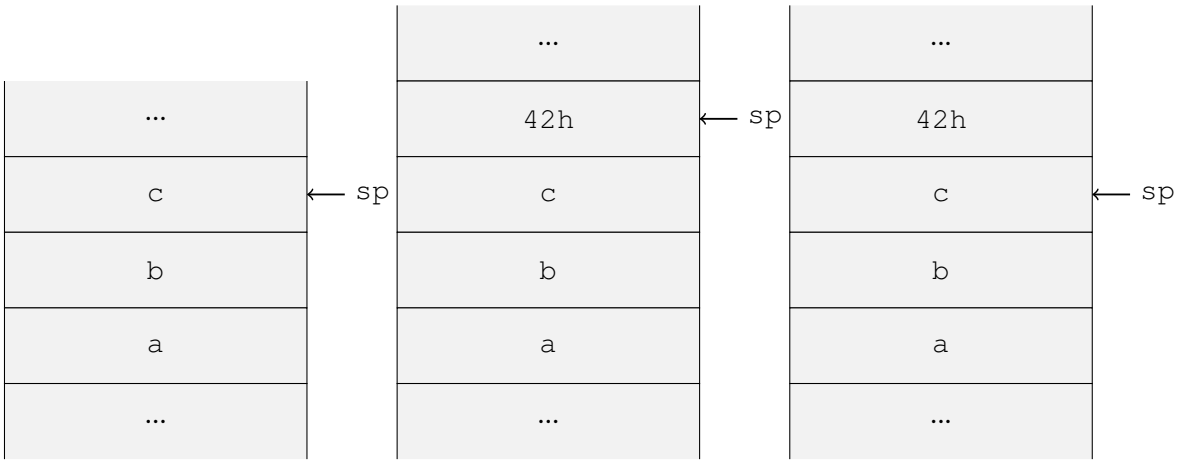
Memory can be allocated statically or dynamically. Static memory allocation uses the stack, is very fast and handled automatically, but space is limited. Dynamic memory allocation uses the heap, is slightly slower than static memory allocation and requires the programmer to manage memory, but has no limits on size.

Stack

The stack is a small area of memory (depending on the compiler and OS between 1 and 8 MiB), allocated for every individual thread. This memory is used by functions as a scratch pad to temporarily store information, local variables, and some housekeeping information. Stacks can be divided further into stack frames, with every function creating its own stack frame on top of the caller's stack frame. We discuss how this works in detail and what data is stored on the stack in Chapter 2.1.3.

Memory management on the stack is inherently simple, thanks to `rsp`: to allocate memory, `rsp` simply needs to be decreased[3] by the required amount. Conversely, to de-allocate

[3]The stack grows towards lower addresses. Therefore, if data is stored in newly allocated memory, it is stored at lower addresses than older data.



(a) Before `push` (b) After `push` (c) After `pop`

Figure 2.1: Change of the stack pointer by `push` and `pop` instructions.[4]

memory, `rsp` is increased. This implies that data on the stack is not erased upon deallocation and remains in memory until it is overwritten.

The two most important instructions that implicitly use the stack for storing and retrieving data, namely `push` and `pop`, always do so from the top, i.e., the lowest address, and automatically change `rsp` accordingly: when a value is pushed on the stack, `rsp` is automatically decreased by 8 bytes. Conversely, when a value is popped off the stack, `rsp` is automatically increased by 8 bytes. `push` and `pop` instructions can therefore also be seen as a combination of a `mov` instruction, loading data `rsp` points to into a target 64-bit register, and the according increase or decrease of `rsp` by 8 bytes (see Listing 2.3 and Listing 2.4).

Listing 2.3: `push` equivalent

```
push rax
; equivalent to:
sub rsp, 8
mov [rsp], rax
```

Listing 2.4: `pop` equivalent

```
pop rax
; equivalent to:
mov rax, [rsp]
add rsp, 8
```

Figure 2.1 shows how the stack and `rsp` change when value `42h` is pushed and then popped into `rax`. Figure 2.1a shows the stack in its original state. Figure 2.1b after the push instruction: `42h` is on top of the stack and `rsp` has been adjusted automatically to point to it. Figure 2.1c shows the stack after the `pop` instruction: `42h` is still on the stack, but it has also been copied into `rax` (not shown) and `rsp` has been adjusted.

[4]Please note that in all figures the stack grows upwards, towards lower addresses i.e., towards address `0`.

Heap

As opposed to the stack, the heap does not have a fixed size. In fact, it can grow almost arbitrarily large. This is why it is usually used for data too large for the stack or when the programmer does not know how much memory will be required. The downside is that the heap is not managed automatically by the hardware in the background, like the stack is. Allocation, de-allocation, and heap management is done in software by the operating system.

Stack space allocation is done automatically in C. Space is reserved with the declaration of local variables and freed when the variable goes out of scope. Space on the heap, however, is always allocated explicitly by using keywords (`malloc` or `new`). This space has also to be de-allocated by the programmer explicitly using `free` or `delete`, as it otherwise leads to a *memory leak*[5].

2.1.3 Function Calls

Normally, program flow is strictly sequential. The instruction at `rip` is read, `rip` is moved to the next instruction, and the previously read instruction is executed:

1. Read the instruction at `rip`.
2. Increase `rip` by n bytes, n being the length of the instruction. This makes sure `rip` points to the subsequent instruction.
3. Execute the instruction read in Step 1.
4. Go to Step 1.

In essence, there are two ways that interrupt this linear program-flow: jumps and calls. A `jmp` instruction simply updates `rip`[6]. A function call, initiated using the `call` instruction, however, requires the program flow to return to the calling function, the *caller*, once the called function, the *callee*, has finished. This implies that i.) the *return address*, i.e., the address where the program has to continue once the callee has finished has to be stored, ii.) information about the current stack frame has to be stored, and iii.) parameters have to be passed to the callee. The following steps show how a function call works conceptually:

1. The caller sets up the registers and stack with the parameters that need to be passed in accordance with the respective calling convention (discussed in the next paragraph).

[5] A memory leak occurs when memory is allocated but not de-allocated. C / C++ do not automatically free unreferenced objects using, e.g., a garbage collector.

[6] For example, `jmp rax` would be equivalent to `mov rip, rax`. However, keep in mind that `rip` can only be changed by certain instructions

2. The `call` instruction pushes the return address on the stack.
3. The `call` instruction sets `rip` to the target address.
4. The callee pushes `rbp` on the stack and sets `rbp` to `rsp`, creating a new stack frame for the callee, which sits on top of the caller's stack frame.
5. The callee reserves some space on the stack by decreasing `rsp`.
6. The remaining instructions in the callee execute.
7. The callee frees the memory it occupies on the stack by setting `rsp` to `rbp`.
8. The callee pops into `rbp`, restoring the original value of `rbp` which was stored in Step 4.
9. The `ret` instruction loads the value `rsp` points to into `rip`. This is the return address which was previously stored in Step 2. `ret` also increases `rsp` by 8 bytes, mimicking a `pop rip` instruction.[7]
10. If parameters were passed on the stack, the callee cleans up the stack by increasing `rsp` again. introduce that too

Steps 4 and 5 are often called *function prologue*, and steps 7 and 8 are called *function epilogue*. Listing 2.5 shows typical function pro- and epilogues[8]. If the base pointer is omitted, i.e., if `rsp` is used to access variables instead of `rbp`, Steps 4. and 8. are skipped. This speeds up function calls, because there are fewer instructions to execute, and frees up a register.

Listing 2.5: Function prologue and epilogue

```
push    rbp             ; Step 4
mov     rbp, rsp        ; Step 4
sub     rsp, 42h        ; Step 5
...
mov     rsp, rbp        ; Step 7
pop     rbp             ; Step 8
ret                     ; Step 9
```

In Step 1 we mentioned that parameters are set up in accordance to the calling conventions. On x86-64, there are two main calling conventions, which are very similar to each

[7]However, since `rip` is not a general purpose register, it cannot be used in conjunction with a `pop` instruction.

[8]Note that Steps 7 and 8 could be condensed by using the `leave` instruction, which executes the same instructions shown in Step 7 and 8 and is therefore semantically equivalent.

other. The *Microsoft Calling Convention*, followed by 64-bit Windows platforms, passes the first four arguments to a function through registers `rcx`, `rdx`, `r8`, and `r9`. Additional arguments are passed using the stack. The *System V AMD64 ABI*, followed, among others, by Linux, passes the first six arguments to a function through registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. Additional arguments are passed using the stack. This is a big difference compared to x86, where many different calling conventions exist, and parameters are almost exclusively passed on the stack.

Consider the function call in Listing 2.6, which follows the Microsoft Calling Convention: lines 1 to 4 load arguments in the appropriate registers. Line 5 pushes the fifth argument, which is in `rax`, on the stack. Line 6 executes the actual call, in this case to a custom function called `myfunc`, which takes five arguments.

Listing 2.6: Function call

```
1 mov     rcx, [rbp-40h]  ; Argument 1
2 mov     rdx, [rbp-38h]  ; Argument 2
3 mov     r8, 1           ; Argument 3
4 mov     r9, 100h        ; Argument 4
5 push    rax             ; Argument 5
6 call    myfunc
```

Next, we show the stack frame before and after the function call. We assume that the caller's name is `foo`. Figure 2.2 shows the stack frame for function `foo`. It contains the return address necessary so `foo` can return to its caller, and the caller's frame pointer, as well as three local variables. `rsp` points to the top of the stack, `rbp` to the saved base pointer. Figure 2.3 shows the stack after `foo` called function `myfunc` and `myfunc` allocating space for a local variable. The saved base pointer in the stack frame of `myFunc` belongs to function `foo`, the saved base pointer in the stack frame of `foo` belongs to `foo`'s caller.

Please note that, for brevity, we had to slightly simplify some concepts. For example, the Microsoft Calling Convention uses a memory area of at least 32 byte called *register parameter area* or *home space* between stack frames, which can be used to spill parameters [103]. The System V AMD64 ABI defines a *red zone*, a 128 byte memory area above `rsp` that can be used without moving `rsp` [100] (Chapter 3.2.2). Those details are, however, not relevant to the understanding of the topics.

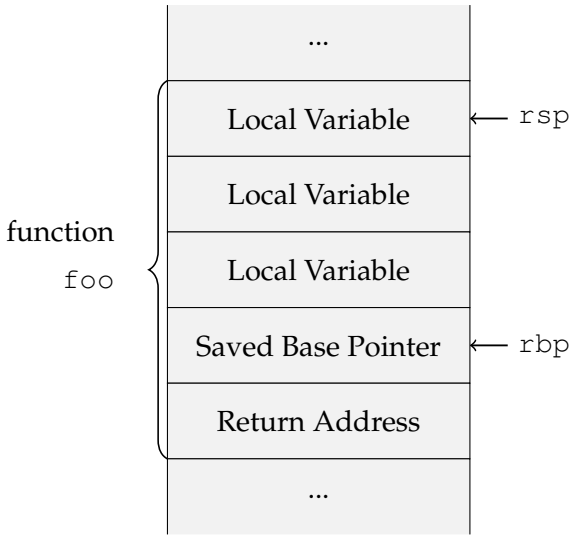


Figure 2.2: Stack with one stack frame

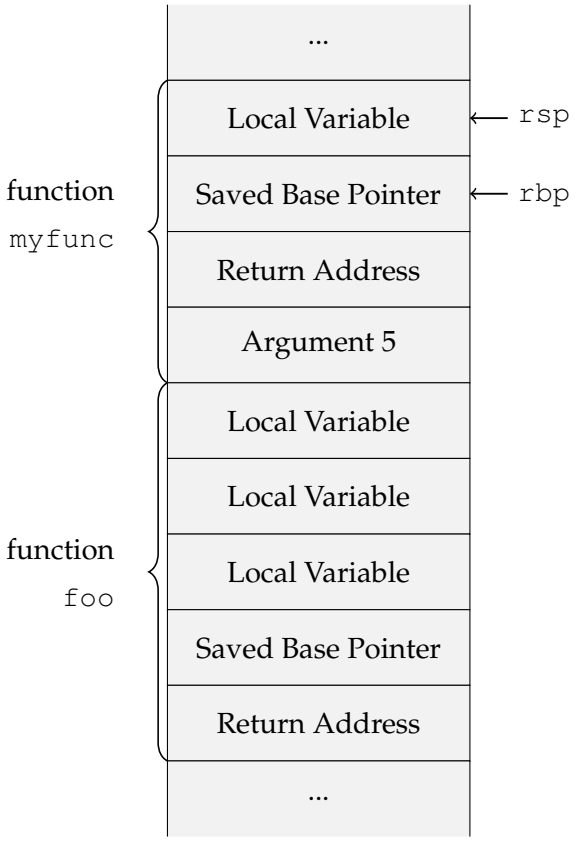


Figure 2.3: Stack with two stack frames

2.2 Bugs and Vulnerabilities

In this chapter we introduce three commonly exploited low-level memory vulnerabilities, but first we define the terms *vulnerability* and *exploit*.

Definition 1. *A vulnerability is "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [145].*

An *exploit* is input that triggers a vulnerability to achieve a certain goal. This can range from crashing the program (denial-of-service, or DoS, attack), to manipulating internal data to change the outcome of an operation, to elevating one's privileges (privilege escalation attack), to arbitrary code execution. Arbitrary code execution is usually the goal of an attacker, as this gives her the most power. It is, however, also the most difficult one to achieve and, depending on the vulnerability, may not be possible at all. This holds, of course, true for all goals, i.e., it is possible that a vulnerability exists, but cannot be exploited to achieve any of the goals stated above.

In the remainder of this chapter, we discuss three widely exploited memory corruption vulnerabilities. Classic *buffer overflows*, most prevalent until about 2010; *use-after-free*, currently widely exploited; and *type confusion*, which is becoming an increasingly large threat. A buffer overflow is a *spatial* memory error, where an out-of-bounds-pointer is dereferenced, while a use-after-free vulnerability is referred to as *temporal* memory error, because a pointer that points to an object which does not exist anymore is dereferenced. These bugs are manifestations of the fact that C/C++ are not *memory safe*, a property commonly defined as not committing spatial or temporal memory errors. Type confusion vulnerabilities exist because C/C++ are not *type safe*. Further, type unsafety results in memory unsafety. We show examples of typical bugs that cause such vulnerabilities as well as techniques to exploit them.

2.2.1 Buffer Overflow

Until about 2010, buffer overflows were the most widely exploited memory-corruption vulnerability accounting for almost two thirds of all exploited vulnerabilities, at least in Microsoft products [10]. They are relatively simple to exploit, especially when compared to use-after-free vulnerabilities, which we introduce in Chapter 2.2.2. Conceptually, a buffer overflow is a simple vulnerability: a fixed-size buffer is filled with more data than it can hold, which causes adjacent memory to be overwritten. This chapter covers the underlying problem of buffer overflows and how they are exploited.

Underlying Problem

We identify two major issues. The first issue is that control-flow data is sometimes stored in the same memory area with other, potentially user-controlled data and not protected in any way. The second issue is that C has no automatic bounds checks, which makes buffer overflows possible in the first place.

Storage of Control-Flow Data. Oftentimes, an address the program uses later for determining control-flow is stored in writeable memory, i.e., on the stack or the heap. Remember, for example, how function calls work: first, parameters to the function are set up. Then, the actual `call` instruction executes and pushes the return address, i.e., the address of the instruction after the `call` instruction, on the stack. Then program execution continues at the call address. When the callee has finished, the previously stored return address is used to return to the caller, where program execution continues. This means that while the callee executes, an address that will be used later to control the program flow is stored on the stack together with other data.

No Implicit Bounds Checks. C heavily relies on arrays if several elements of the same type need to be stored. A typical example is the data type string, which does not exist in C. Instead, strings are stored in an array of characters, i.e., as a series of characters terminated by the string termination character `'\0'`. The size of an array is static, and no meta-information, such as its length, is stored with it[9]. Furthermore, during compilation all information about variables and their respective sizes is lost. This is due to the fact that variable declarations in C are compiled to a single memory allocation, i.e., moving `rsp` to allocate space for them in the function prologue. Listing 2.7 shows C code, Listing 2.8 its ASM counterpart. Note how only one large block of memory, which is large enough to hold all variables, is allocated (the `long long` type is 8 bytes long, `char` is 1 byte long). Accessing variables works simply through an offset of `rbp` or `rsp`, if base pointer omission is used as variables per se do not exist, only memory locations.

Listing 2.7: Variable declaration

```
1 int main() {
2         long long i, j, k, l;
3         char buffer[32];
4         ...
```

[9]During the design of C there were discussions whether a string terminator should be used or if the length of the array should be stored in the first byte. Ultimately, it was decided to use a terminating character, as this more closely resembled C's predecessor, B [134]

Listing 2.8: Space allocation

```
1 sub     rsp, 40h              ; 40h = 64
```

Both of these issues combined cause the problem that it is not possible to implicitly check whether accesses to an array are within its bounds. While it is to some extent possible to extract information about variables, e.g., from debugging symbols, forcing bounds checks on every array access has a negative effect on performance, one of the main benefits of C / C++. Therefore, it is the programmers' responsibility to explicitly implement correct bounds checks. If they neglect this, it is possible that memory accesses are out of bounds. Listing 2.9 shows an example of this. This simple program creates a buffer and then uses a loop to initialize its elements to the character 'A'. `buffer` contains 32 elements (line 2), but the loop (line 4) will execute 33 times[10]. In the last iteration the program overwrites data which is not part of `buffer`, more precisely, it overwrites the byte following `buffer`. Depending on what data is stored at this location, the program might crash.

Listing 2.9: Out of bounds access in an array

```
1 main() {
2         char buffer[32];
3
4         for (int i = 0; i <= 32; i++ ){
5                 buffer[i] = 'A';
6         }
7 }
```

Exploiting Buffer Overflows

In Chapter 2.1.3 we reviewed the concept of function calls, which store the return address on the stack. If a buffer overflow affects a buffer located on the stack it may be possible to overwrite a return address. By carefully crafting an input that triggers the vulnerability, an attacker may therefore overwrite the return address with an arbitrary value, hijacking the program flow. This attack is know widely known as *stack smashing* [3]. Listing 2.10 shows code containing a buffer overflow that can be exploited by an attacker to hijack the control-flow.

[10]This is known as an off-by-one-error.

Listing 2.10: A buffer overflow vulnerability

```
1 int vuln() {
2         char buffer[24];
3         long long length = 0;
4         gets(buffer);
5         length = strlen(buffer);
6         return length;
7 }
```

This function reads user input and returns the input's length. Line 2 declares a variable called `buffer`, which can store 24 bytes. The actual usable size is 23 bytes, because the string termination character (`\0`) requires another byte. Line 3 declares an integer variable called `length` and initializes it to `0`. Line 4 reads user input from the keyboard into variable `buffer` using library function `gets`. Line 5 computes the length of `buffer` using library function `strlen` and stores it in `length`. Line 6 returns the value of `length` to the caller.

In this toy example, line 4 introduces a buffer overflow vulnerability. In this line, user input is read into `buffer`, a variable with a fixed length of 24. Therefore, if the user enters more than 23 characters (the string terminator is appended automatically after hitting enter), contents in adjacent memory, in this case the saved base pointer and the return address, are overwritten. Figure 2.4a and Figure 2.4b show the stackframe of function `vuln` before and after the overflow[11]. Note that the program still expects the return address at the same location, therefore, it can be controlled by the attacker.

Assume a malicious user tries to actively exploit the vulnerability, entering the following string: `AAAAAAAAAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD` The stack will look like shown in Figure 2.5. When function `vuln` has finished and the function prologue runs (see Chapter 2.1.3), Step 7 works correctly, but Steps 8 and 9 load user-controlled data into `rbp` and `rip`. `rip` contains `4444444444444444h` (the ASCII value of 'D', which is `44h`, repeated 8 times), an address where likely no code is mapped, causing an access violation which leads to a crash of the program. However, since the user can enter arbitrary data, she can craft a more malicious exploit, shown in Figure 2.6. Here, the user injects shellcode and overwrites the return address with the start address of the buffer, in this case `18FF00h`. When the program executes the `ret` instruction, `18FF00h` is loaded into `rip`, which causes the shellcode to be executed.

This is just one possible exploit structure. It has the disadvantage, that the attacker has only 40 bytes of space available for her shellcode. Another approach would be, for example,

[11]This is a generic visualization and different compilers will produce different layouts.

Listing 2.11: Fixed buffer overflow

```
1 int vuln() {
2         long long length = 0;
3         char buffer[16];
4         fgets(buffer, sizeof(buffer), stdin);
5         length = strlen(buffer);
6         return length;
7 }
```

to place the shellcode beyond the current stack frame, resulting in a layout as shown in Figure 2.7. If the program allocates space on the heap it may also be possible to place the shellcode there.

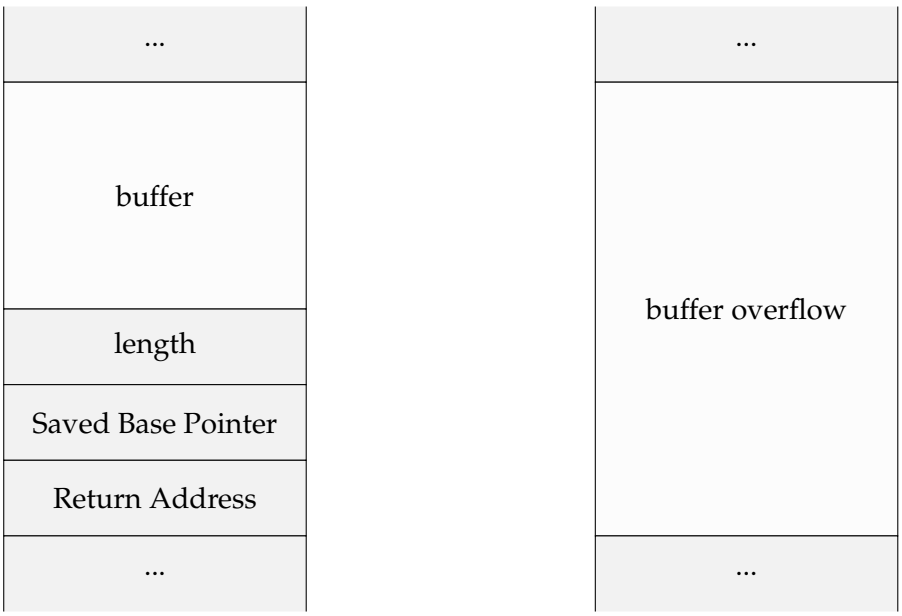
Lastly, Listing 2.11 shows the code snippet from Listing 2.10 with the buffer overflow bug fixed. Instead of reading user input using `gets`, `fgets` is used. It takes two additional parameters, the maximum size that should be read, which is set to the size of the buffer using the `sizeof` function, and where data should be read from. In this example we use `stdin`, which is the keyboard.

2.2.2 Use-After-Free

Due to the popularity of buffer overflow exploits, awareness for this bug has increased and researchers proposed many effective mitigation techniques, which we introduce in Chapter 2.3. Due to these steps, the security community has noticed a decrease in buffer overflow exploits. Instead, attackers now often target use-after-free vulnerabilities, which are an attractive target due to the fact that, unlike buffer overflows, no mitigation techniques against such attacks have been widely deployed at this point. In 2013, use-after-free exploits accounted for almost 50% of exploited Microsoft CVEs, while only about 25% were buffer overflows [10].

Underlying Problem

In essence, a use-after-free vulnerability is caused by a programming bug where an object that has been freed at time t is accessed after t. A pointer to such a freed object is called a *dangling pointer*. If a dangling pointer is dereferenced it results in unspecified behaviour. Oftentimes a program will simply crash, because the allocated heap memory has since been



(a) Before buffer overflow

(b) After buffer overflow

Figure 2.4: Stack before and after a buffer overflow.

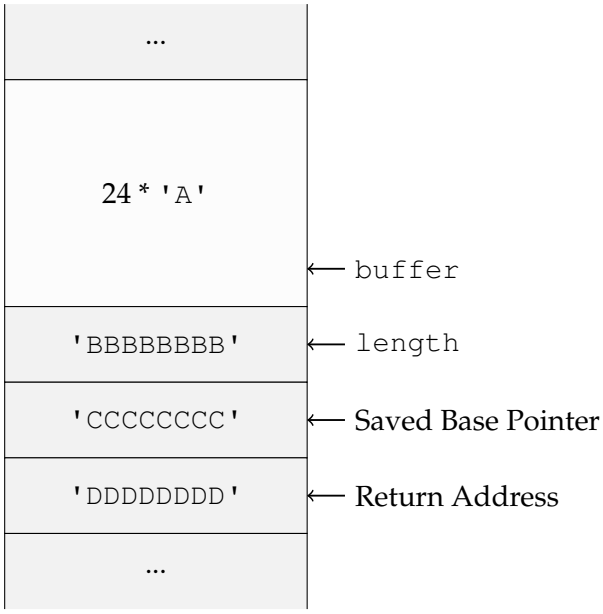


Figure 2.5: Buffer overflow

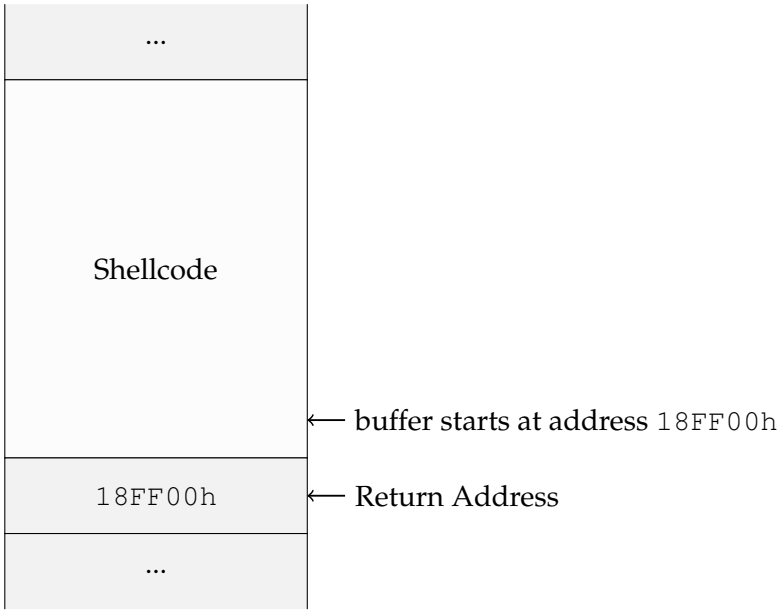


Figure 2.6: Exploited buffer overflow with shellcode before the return address

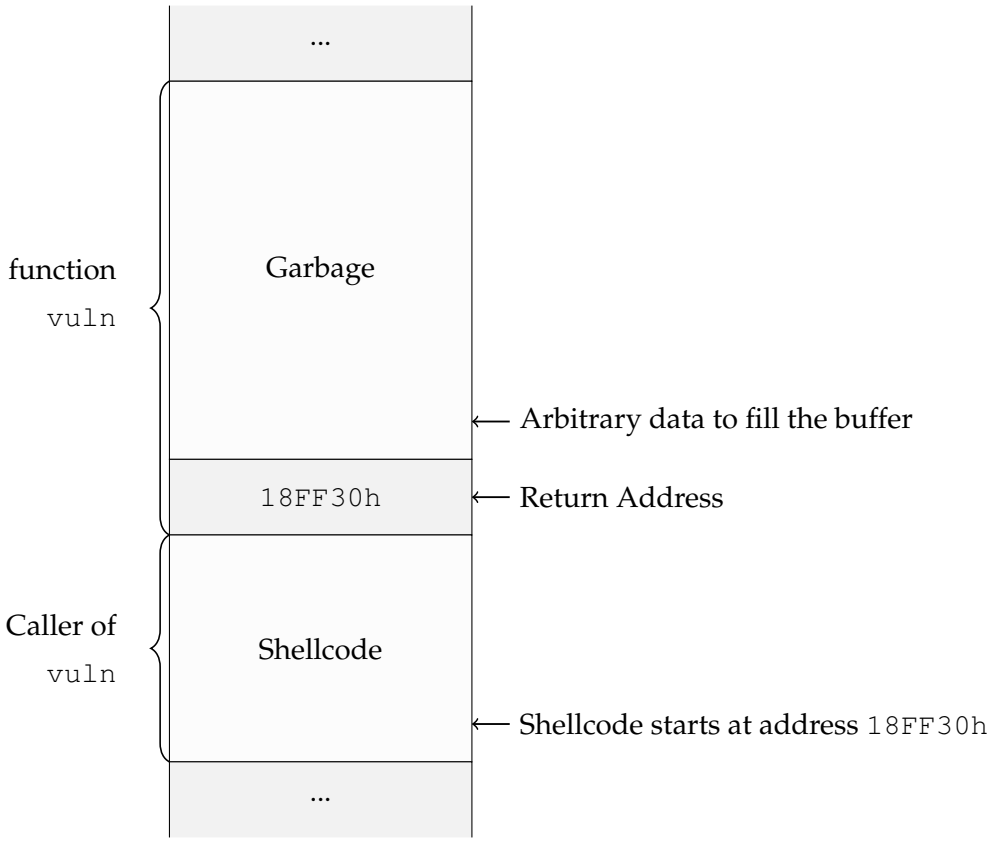


Figure 2.7: Exploited buffer overflow with shellcode after the return address

Listing 2.12: Use-after-free bug #501572 in openssl

```
1 dtls1_hm_fragment_free(frag);
2 pitem_free(item);
3
4 if (al==0)
5         {
6         *ok = 1;
7         return frag->msg_header.frag_len;
8         }
```

reused to store a different object or other data. In other cases, the program might continue running correctly.

Listing 2.12 shows an example of a real use-after-free bug[12]. In line 1, object `frag` is freed. However, in line 7, `frag` is dereferenced if `al` is zero.

Use-after-free vulnerabilities are most severe if objects containing a vtable are affected. A vtable is used to implement dynamic binding, also known as late binding. We use the code in Listing 2.13 as an example: a base class `Animal` has two derived classes: `Cat` and `Dog`. All three classes have a function called `identify`, which prints the name of the respective class, and a function `makenoise` which prints the noise the animal typically makes. In the main function, an `Animal` pointer is created, which is randomly either of type `Cat` or `Dog`[13]. Then, on this object function `makenoise` is called. The output of this code fragment is either "Meow" or "Woof". Due to late binding, the program can determine the object the pointer points to at runtime and invoke the correct function, despite the pointer being of type `Animal`.

Late binding is implemented using *vtables*. The concrete implementation may differ between compilers, but Figure 2.8 shows the abstracted concept, based on the code in Listing 2.13. When a new object is created, the required number of bytes to store it are allocated. In addition to that, space for a pointer is allocated. This pointer points to the *virtual function table*, or *vtable*, and is called *vpointer*. The vtable is stored somewhere in memory[14] and

[12] `https://bugzilla.redhat.com/attachment.cgi?id=344671&action=diff`

[13] The random assignment ensures that the compiler cannot statically detect which function should be invoked, thereby forcing it to create a vtable and the necessary pointers. Otherwise the compiler might optimize these structures away.

[14] The location depends on the compiler.

Listing 2.13: Virtual functions

```
class Animal {
        public:
                unsigned int number_of_legs;
                unsigned int weight;
                virtual void identify()
                        {cout << "I'm␣a␣generic␣animal\n";}
                virtual void makenoise()
                        {cout << "Generic␣noise\n";}
        };

class Cat: public Animal {
        public:
                void identify()
                        {cout << "I'm␣a␣cat\n";}
                void makenoise()
                        {cout << "Meow\n";}
        };

class Dog: public Animal {
        public:
                void identify()
                        {cout << "I'm␣a␣dog\n";}
                void makenoise()
                        {cout << "Woof\n";}
        };

main() {
        Animal* a;
        if(rand() % 20 < 10){
                a = new Cat;
        } else {
                a = new Dog;
        }
        a->makenoise();
}
```

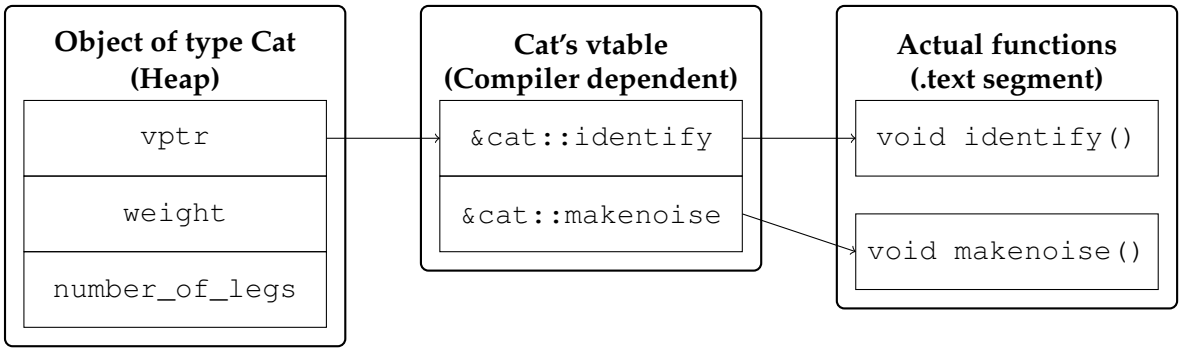


Figure 2.8: Object with virtual functions

Listing 2.14: Virtual function dispatch

```
1 mov     rcx, ...         ; load object in rcx
2 mov     rax, [rcx]       ; load vtable in rax
3 mov     rdx, [rax + 8]   ; load second vtable entry in rdx
4 call    rdx              ; call virtual function
```

contains the addresses of the actual functions, which are, as all static code, in the `.text` or `.code` segment.

Listing 2.14 shows how virtual function dispatching works on the assembler code level. We assume in this example that a `Cat` object has been created. Line 1 loads a pointer to the object in `rcx`. In line 2, `rcx` is dereferenced and the result is stored in `rax`. As the memory layout in Figure 2.8 shows, the first 8 bytes of a `Cat` object are occupied by its `vptr`, a pointer to the object's vtable. Therefore, the second line loads the address of the vtable in `rax`. In line 3, the second entry of the vtable is retrieved and stored in `rdx`. Therefore, `rdx` now contains the address of `cat::makenoise()`. Lastly, in line 4, an indirect call to `rdx` invokes the function.

Exploiting Use-After-Free

The behaviour described in the previous chapter creates a large attack surface, as an attacker can, for example, overwrite the original vptr of a deallocated object, making it point to a fake injected vtable, which allows arbitrary control-flow changes. The difficulty in exploiting a use-after-free vulnerability lies within the fact that the attacker needs the ability to reliably re-allocate the freed memory before it is dereferenced again, as this allows her to overwrite control-flow data. Figure 2.9 shows these steps visually.

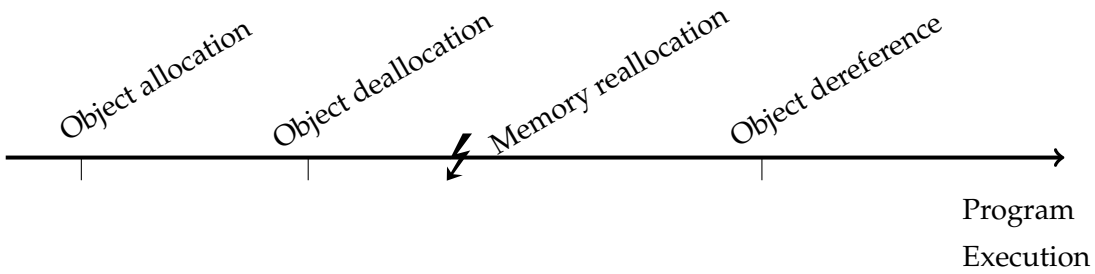


Figure 2.9: Use after free exploitation timeline.

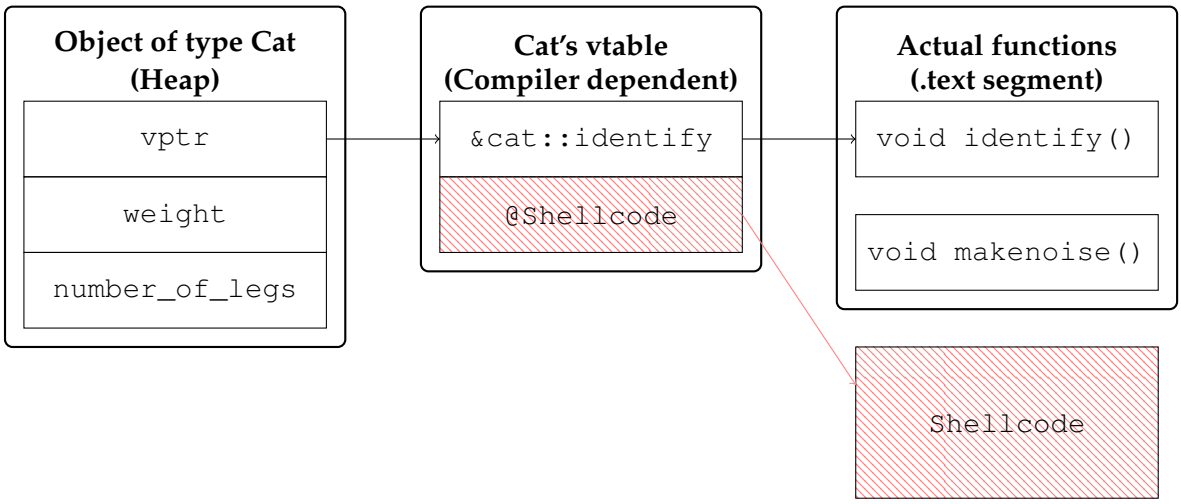


Figure 2.10: A vtable corruption attack. Red marks attacker-controlled data.

If the attacker is able to reliably re-allocate the memory area, for example, by using techniques such as Heap Feng Shui [153], she has several ways of exploiting a user-after-free vulnerability. We assume that a program invokes `makenoise()` on a `Cat` object, i.e., dereferences the second entry of the corresponding vtable.

vtable Corruption Technically, this is just a general attack on vtables and does not rely on a use-after-free vulnerability. However, since it is a realistic threat, we discuss it in this chapter. A vtable corruption attack attempts to overwrite one or several entries in a vtable. Since a vtable usually exists only once in memory and all objects of that type have a reference to it, whenever such an object accesses the vtable it accesses corrupted data, giving the attacker control over the program flow. Figure 2.10 shows this in a visual way. The second entry of the vtable is overwritten with the address of shellcode, which is placed somewhere else in memory. This attack requires that vtables are placed in writeable memory.

vtable Injection In this attack, the attacker crafts and injects a data structure similar to a vtable, i.e., an array of pointers. After an object has been freed, the attacker reallocates the memory area the original object has occupied and overwrites the `vptr` with the address of the injected vtable. When a virtual function of that object is invoked, the data from the

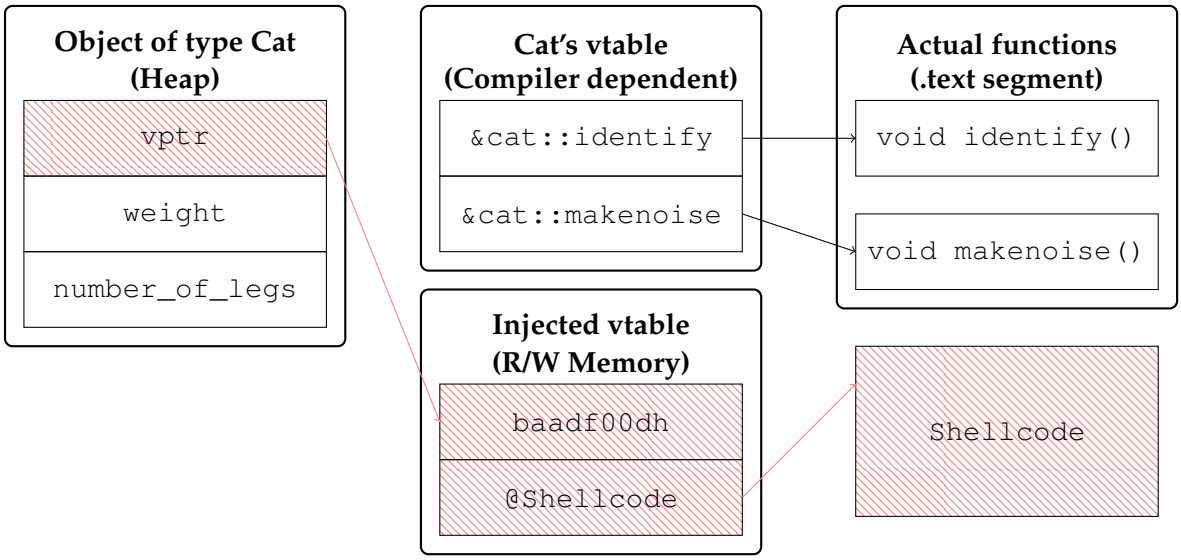


Figure 2.11: A vtable injection attack. Red marks attacker-controlled data.

injected vtable is used. Figure 2.11 shows this visually. The original `vptr` is overwritten, and shellcode and a fake vtable are injected. The first entry of the vtable is irrelevant because only the second one is dereferenced, which points to the injected shellcode.

vtable Reuse Lastly, an attacker may also overwrite the `vptr` with an address pointing to other data which will result in a valid address when interpreted as address. This might be necessary, if a compiler generates additional checks that make sure, all vtables are stored in read-only memory. Note that only the vptr is attacker controlled, but not the data it points to. Also note, that the pointer points at arbitrary data in read-only memory, which can be another vtable, but also any other data. In this scenario the attacker is very lucky as there is data which, when interpreted as address, points to her shellcode. Figure 2.12 depicts this visually.

2.2.3 Type Confusion

Exploits based on type confusion vulnerabilities are becoming an increasingly large threat, especially for programs written in C++. In October 2016, a search for "type confusion" in the common vulnerabilities and exposure database[15] shows 96 results, 84 of which are from 2014 or later. Among affected programs are popular and widely used ones such as Chrome, Firefox, Flash, and PHP.

[15] `https://cve.mitre.org/`

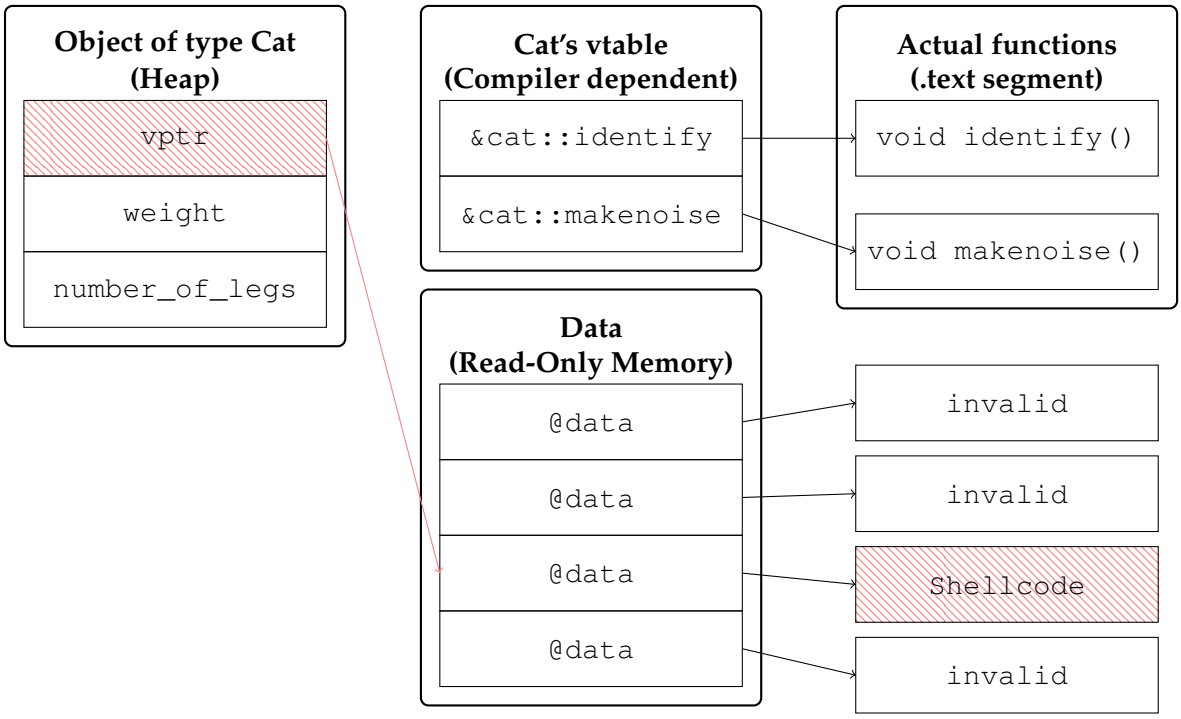


Figure 2.12: A vtable reuse attack. Red marks attacker-controlled data.

Underlying Problem

Generally, type confusion is caused by accessing an object of one type as another type, which C++ allows through *casting*. C++ offers a variety of cast operators but for performance reasons `static_cast` is often preferred. However, as opposed to dynamic casting forced through `dynamic_cast`, static casts cannot guarantee that a cast is not illegal. This can lead to type confusion vulnerabilities, as shown in Listing 2.15. In line 11, an object of type `Animal` and a pointer of the same type, which points to it, are created. In line 12, the pointer is casted to type `Cat`, and in line 13, the member `ruined_objects`, which only type `Cat` has, is accessed. This leads to an out-of-bounds memory write.

Figure 2.13 shows the corresponding memory layouts of `Animal` and `Cat` objects. Figure 2.14 shows how, due to an unsafe cast[16], adjacent data is erroneously interpreted as member variable `ruined_objects` and overwritten.

Exploiting Type Confusion

If the overwritten memory contains control-flow data, such as a vtable or return address, and the attacker can control the data written to it, she can hijack the control-flow. These

[16]More precisely a *downcast*, as a pointer of a base class is casted *down* to a derived class. The opposite operation is called *upcasting* and is generally safe, because all members of the base class must also be present in the derived class. Therefore, it is not possible to access a member that does not exist.

Listing 2.15: Unsafe downcast

```
1  class Animal {
2                  ...
3          };
4
5  class Cat: public Animal {
6          public:
7                  unsigned int ruined_objects;
8          };
9
10 main() {
11         Animal* a = new Animal();
12         Cat* c = static_cast<Cat*>a;
13         c.ruined_objects = 1;
14 }
```

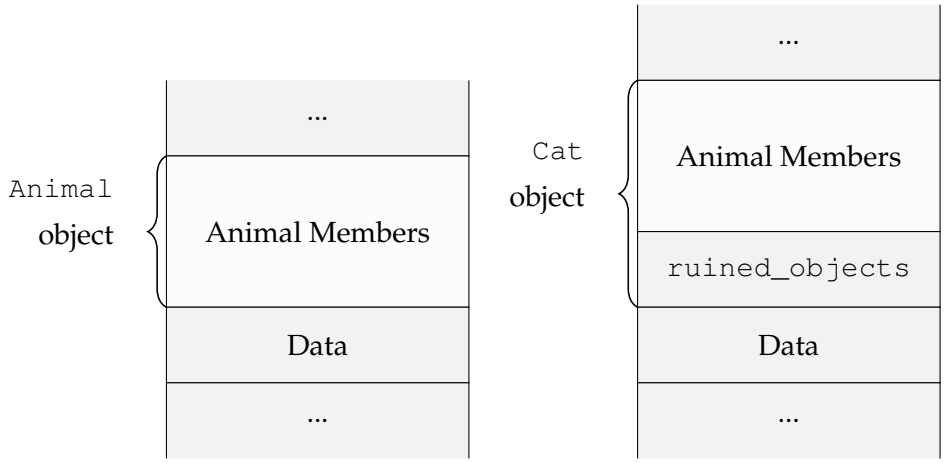


Figure 2.13: Type confusion memory layout

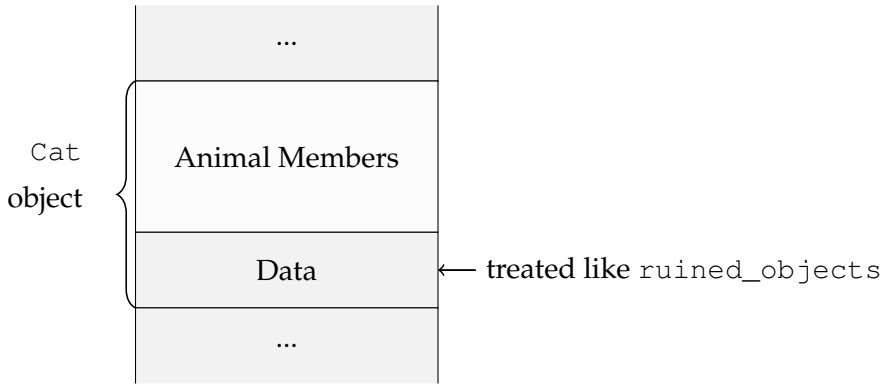


Figure 2.14: Type confusion vulnerability caused by treating an `Animal` object as object of type `Cat`

attack vectors are very similar to buffer overflow exploits and use-after-free exploits, which we described in Chapter 2.2.1 and Chapter 2.2.2, respectively.

2.3 Exploit Mitigation

This chapter presents currently used mitigation techniques, which are designed to prevent exploitation of memory-corrupting vulnerabilities. The most common ones are DEP, ASLR, and stack cookies, which are an integral part of both Windows and Linux security, and have also made their way into most mobile operating systems, despite low-level attacks not being a real threat yet on those platforms. We also introduce various other, less widespread mitigation techniques.

2.3.1 Executable Space Protection

We use executable space protection as a collective term describing techniques that protect against traditional code injection attacks, in which in attacker injects code into a running program and then uses a memory corruption vulnerability to redirect control flow to this code (see Chapter 2.2). Such mitigation techniques prevent injection attacks by enforcing that memory pages that do not contain code, such as heaps and stacks, are mapped as non-executable. There are various names for technologies implementing such protection; on Windows it is called data execution prevention (DEP) [4], for Linux there are W^X [52], PaX [123], and Exec Shield [163]. In this work, we refer to all techniques as DEP.

DEP employs a mix of hardware and software to enforce non-executable pages. It makes use of a processor's NX (No eXecute) bit, marking non-code pages as non-executable by setting bit 63 in their respective page-table entry. `1` means that the contents of the page are not executable, while `0` means that the contents of the page are executable. An attempt to

put the program counter inside a protected page causes a hardware-level exception. On the software side, operating systems need to support DEP so page tables can be configured correctly. Generally, software has to be designed in such a way that it does comply with DEP. This, however, can pose a problem, especially with self-modifying code or dynamically generated code (JIT compiled code), very common in browsers to implement, e.g., Java Script. For such cases, software needs to allocate memory using appropriate APIs, such as `VirtualProtect` or `VirtualAlloc` on Windows, or `mprotect` or `mmap` on Linux. These APIs allow changing the protection level of already allocated pages or allocating memory with a programmer-specified protection level. DEP is an integral part of of today's defences against code injection attacks and supported, among others, by Windows (since XP SP 2), Linux (kernel 2.6.8 or higher), OS X (version 10.4.4 or higher) Android (version 2.3), and iOS (version 5).

DEP is extremely effective and efficient since it is implemented in hardware, and to this day, there are no attacks that exploit a weakness in the concept itself. However, DEP neither fixes the underlying vulnerability, nor does it prevent hijacking of the control-flow. This allows code-reuse attacks, in which an attacker puts together pieces of existing code. We discuss such attacks in depth in Chapter 2.4.

2.3.2 Address Space Layout Randomization

Most attacks rely on the premise that an attacker knows about a program's memory layout, i.e., knows where in memory certain data resides. For example, in Chapter 2.2.1 we have shown an attack where the address of the shellcode was hard-coded. Until about 2005 this premise was true, because programs and libraries were always loaded at the same static address. However, the introduction of ASLR [15, 80, 122] removes this knowledge from the attacker by randomizing the address of the program image, its libraries, heaps, and stacks. Therefore, each time a program is restarted, all addresses will be different from the previous run. Figure 2.15 shows this in a visual way. This forces an attacker to guess memory addresses and a wrong guess usually results in a program crash.

ASLR provides only probabilistic security. Depending on the entropy used for randomization, an attacker may be lucky enough to successfully guess the correct memory address. Under certain circumstances an attacker may also be able to guess repeatedly without causing a program crash. A well known example is the Apache server, where for every connection a new child process is forked. These child processes, however, are not randomized again but share the memory layout with the main process. This gives an attacker the opportunity to try different addresses, until a correct one is found. This is, of course, a very noisy attack and might be detected by an intrusion detection system (IDS). The preferred method

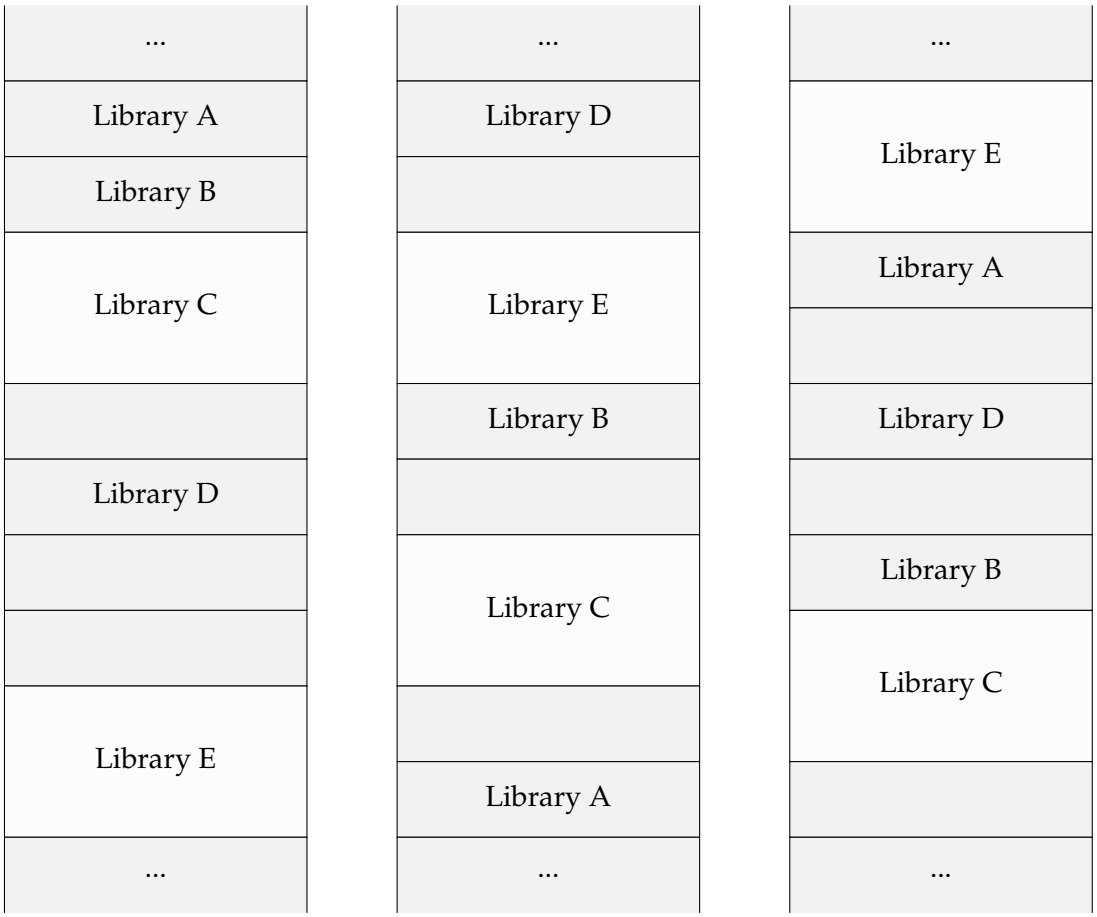


Figure 2.15: Different memory layouts between reboots due to ASLR

of bypassing ASLR and other mitigations that are based on hiding information in memory, however, are information leaks [32, 142, 143].

Another big issue is granularity. ASLR only randomizes the start address of an executable, but not its contents. This means that offsets within the binary stay the same. An attacker can use this knowledge to her advantage: by discovering a single pointer to a known location inside the binary, she can calculate all other addresses inside that binary simply by applying static offsets.

Lastly, ASLR is not necessarily enabled for all binaries of a program. Especially legacy software might not have been linked with the appropriate switches, leaving some binaries unprotected. Another risk is software that introduces and loads 3rd party libraries which are not ASLR enabled. A popular example is Java 6, which loads `msvcr71.dll`, a library that is not relocatable and has not changed since 2006. This library has been used by many universal ASLR DEP bypasses, such as Sayonara [39,83]. We introduce more techniques and discuss all attacks mentioned so far in more detail in Chapter 3.

2.3.3 Stack Canaries

A stack canary [23, 41], also knows as stack cookie, is a pseudo-random number which is inserted on the stack before the saved base pointer and the return address as part of the function prologue[17]. Upon returning from a function, in the function epilogue, the value of the canary is compared to a protected copy and if they do not match, the program aborts. This is very effective because if an attacker exploits a buffer overflow to overwrite a return address, the canary is inevitably overwritten, too. Figure 2.16 shows a stack frame where a canary is protecting the return address.

The implementation of a canary is dependant on the compiler which has to generate code accordingly[18]. As opposed to DEP and ASLR, no hardware or operating system support is required, allowing any program to be compiled with a canary. To increase performance, compilers assess whether a function is vulnerable to buffer overflows and omit stack canaries if this is not the case. For example, a function that has no array variables is not susceptible to buffer overflows and therefore does not require a stack canary.

Stack canaries suffer from several weaknesses. They only protect return addresses, but other code pointers such as entries in vtables are not protected. Furthermore, the canary, like ASLR, offers only probabilistic security, as an attacker may be able to guess the correct value. Lastly, stack cookies share yet another weakness with ASLR, which are memory leaks. If an attacker is able to read the value off the stack, she can incorporate it into her exploit by overwriting the stack cookie with the correct value.

2.3.4 Control-Flow Integrity

Control-Flow Integrity (CFI) [1] is based on the observation that correct program flow can be described by a control-flow graph (CFG). This graph contains all possible paths through a program. Even indirect control-flow transfers are somewhat deterministic, as, e.g., `ret` must return to the caller. CFI uses this knowledge and inserts checks before indirect control-flow transfers that ensure that the target is in line with the CFG. This effectively forces a program to not deviate from the pre-computed paths.

While CFI has seen a large increase in interest by researchers in recent years, it has not yet been widely adopted. Microsoft introduced Control Flow Guard [101] in Windows 8.1 and Visual Studio 2015. Clang supports CFI since version 3.9 [36, 162].

Many different CFI implementations have been proposed. We discuss them along with weaknesses in Chapter 3. Some implementations protect only *forward edges*, i.e., indirect

[17] The position of the canary, i.e., whether it is placed before or after the saved base pointer, may vary between compilers. Also, some compilers may use a different kind of canary, such as a terminator canary, which uses characters that terminate strings instead of a random value.

[18] In Visual C++ the switch is called /GS, in GCC -fstack-protector

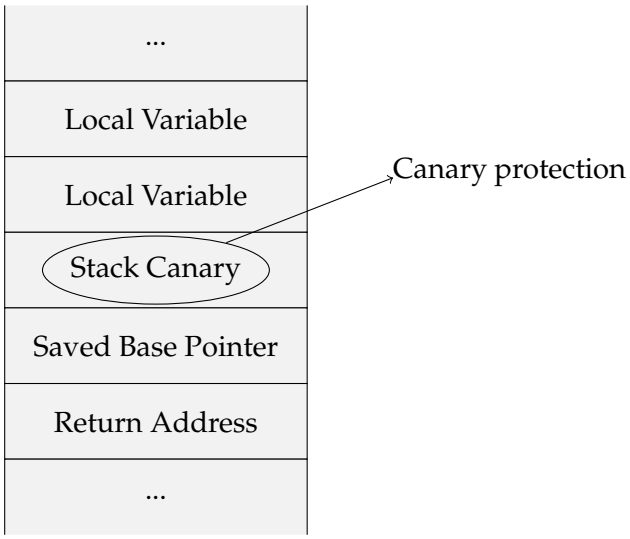


Figure 2.16: Stack canary

`call` and `jmp` instructions, some protect only *backward edges*, i.e., `ret` instructions, and some protect both. In this chapter, we only introduce the general approach, which most implementations rely upon. The biggest differences between various approaches are shortcuts to increase performance and adding additional mitigations to make the implementation more secure.

CFI is usually implemented using labels or lookup tables. In this chapter we describe how implementations using labels typically work. After constructing the CFG, CFI inserts a unique label at all legal destinations as determined by the CFG. Before an indirect control-flow transfer, code is added that checks that the target is in the set of legal targets. Assume, for example, there is a `call rcx` instruction and, using the CFG, CFI determines that the only legal targets of this call are a functions `A` and `B`. Function `A` is assigned the label `10`, function `B` is assigned the label `20`. CFI inserts instrumentation code just before the call, which checks whether `rcx` points to either `10` or `20`. If this is not the case, an illegal target would be called, which indicates an ongoing attack. If the target is legal, the jump is taken and an offset is added, to ensure that the label is skipped.

2.3.5 Variable Reordering

Some compilers make sure that stack variables are ordered in such a way that arrays can only overwrite other arrays but not other non-array variables. To achieve this, arrays are located together below any other variables. Listing 2.16 shows some variable declarations, which are compiled to two different stack layouts, shown in Figure 2.17. Without variable reordering, as shown in Figure 2.17a, either buffer can overwrite `isAdmin`, a variable that

controls which rights a user has. In Figure 2.17b, this is not possible, because an overflow in either one of the arrays cannot overwrite non-array local variables.

SafeStack [35] takes this idea one step further and completely separates return-addresses, spilled registers, and variables whose access it deems safe from other data, notably arrays, using a separate area of memory. SafeStack is part of CPI [91] and currently only available in LLVM, with plans to port it to GCC. We discuss CPI in more detail in Chapter 3.1.

```
1 int main() {
2         int i;
3         char usr_in[16];
4         char usr_out[16];
5         int len;
6         bool isAdmin;
7         ...
8 }
```

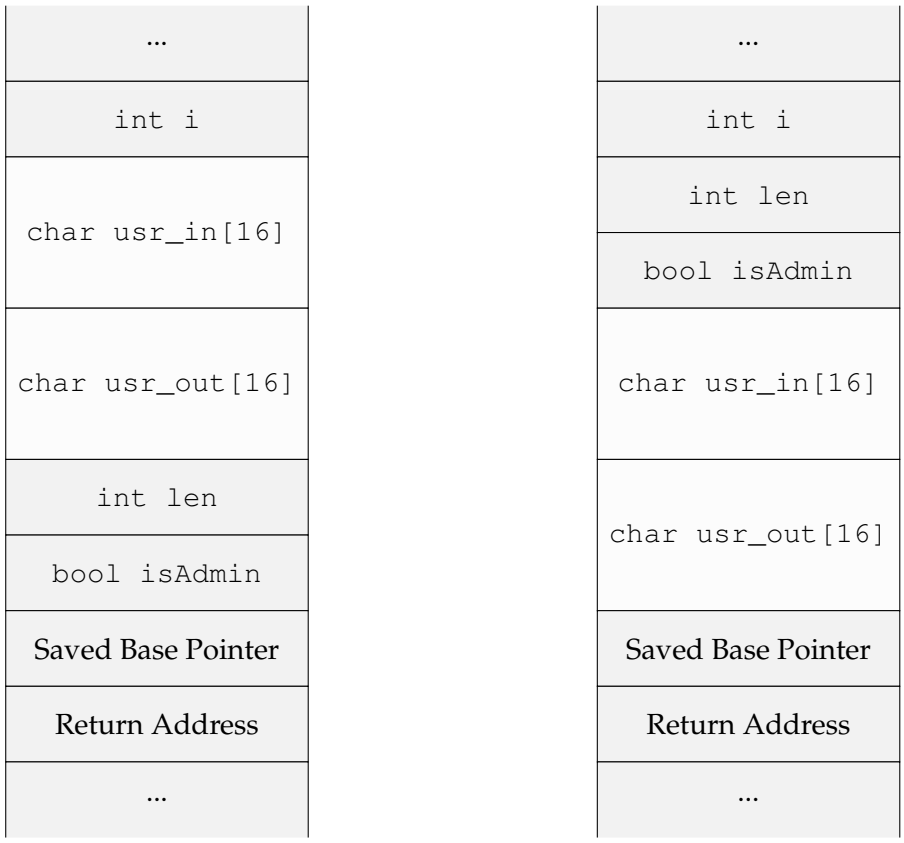
Listing 2.16: C code to show variable reordering

2.4 Code-Reuse Attacks

Code-reuse attacks were born out of the necessity to find a new attack vector after DEP prohibited executing injected code. In code-reuse attacks, no new code is introduced to the attacked system and instead, fragments of existing code, dubbed *gadgets*, are put together in a malicious way. These gadgets can be taken from the main binary itself or any of its dependencies.

In this chapter, we describe how code-reuse attacks work and some of the different manifestations that exist. First, in Chapter 2.4.1 and Chapter 2.4.2, we introduce techniques that allow putting gadgets together in a way so they execute consecutively. Then, in Chapter 2.4.3, we discuss gadgets in more detail, show special kinds of gadgets and introduce an interesting property of the x86-64 architecture that allows discovering more gadgets.

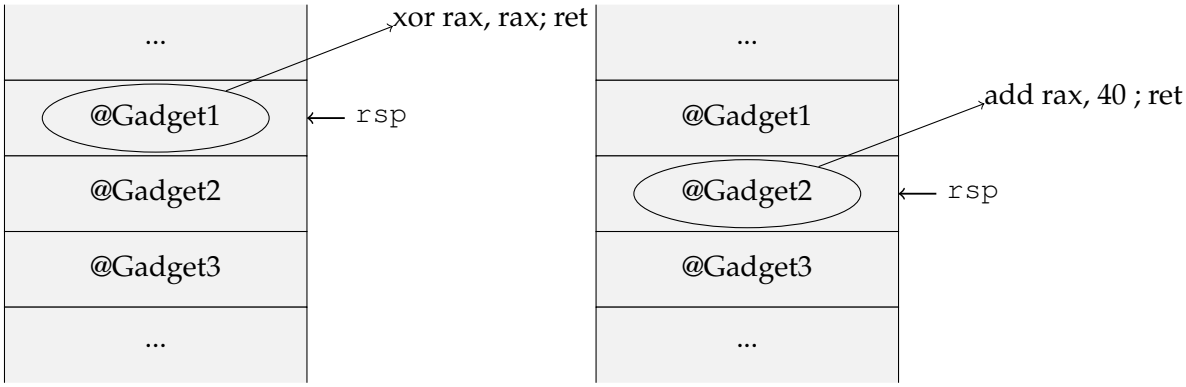
Currently, several techniques that allow putting gadgets together in such a way that they execute consecutively, exist. Often, they can be combined. The most widespread one is called return-oriented programming (ROP) [135, 144], which we introduce first. Another technique called jump-oriented programming (JOP) [18, 28, 107] is barely used, but to facilitate a deeper comprehension of code-reuse-attacks, we still introduce it. There are many derivatives, which are, however, not used in practice, and we refer the interested reader



(a) No variable reordering

(b) With variable reordering

Figure 2.17: Different stack layout due to variable reordering



(a) ROP 1 (b) ROP 1

Figure 2.18: ROP

to the original papers: string-oriented programming (SOP) [127], sigreturn-oriented programming (SROP) [19], data-oriented programming (DOP) [81], crash-resistant oriented programming (CROP) [68], or printf-oriented programming [25].

2.4.1 Return-Oriented Programming

The `ret` instruction is basically a `pop rip` instruction, meaning that if an attacker can control the memory area around `rsp`, she has control over the program execution. This allows her to execute code fragments from arbitrary memory locations whose addresses are placed on the stack. Injecting the addresses in a program's memory space is usually not a problem, since all useful programs take some form of input. Consider the example shown in Figure 2.18a, assuming the next instruction executed is a `ret` instruction: `ret` loads the value `rsp` points to into `rip`, sets `rsp` to the next value on the stack, and program execution continues at the abstract address `@Gadget1`. Disassembling at this address reveals an `xor rax, rax`[19] and then another `ret` instruction. The CPU will execute the `xor` instruction and then arrive at the `ret`. At this point, shown in Figure 2.18b, `rsp` points to the abstract address `@Gadget2`. The `ret` instructions is executed, loading `@Gadget2` into `rip` and increases `rsp` so it points to `@Gadget3`. Then, `@Gadget2`, which ponts to an `add rax, 40` and a `ret` instruction sequence, executes. This example shows how gadgets ending with a `ret` can be put together so they execute consecutively. For the purpose of ROP, `rsp` can be considered another program counter, as it is responsible for the control flow: `rsp` moves from gadget to gadget, and `rip` executes the actual instructions of a gadget.

This very simple example initializes register `rax` to `40h` without introducing and executing any new code. All code that is executed is taken from the main executable or any of its

[19] `xor`'ing a register with itself basically zeroes that register and is faster and requires fewer bytes to store than the alternatives (such as `mov rax, 0` or `sub rax, rax`).

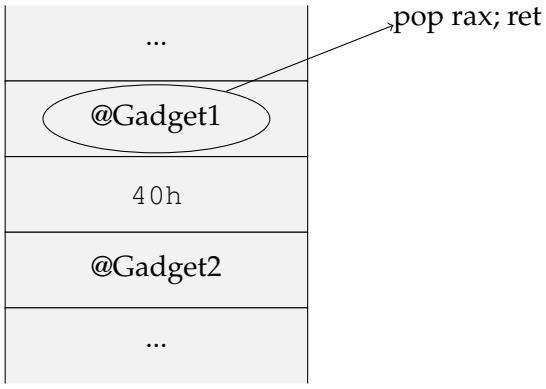


Figure 2.19: ROP with data intertwined

dependencies. Another way of achieving the same goal is to intertwine data with addresses of gadgets, and then use gadgets to load that data into the appropriate register, as shown in Figure 2.19. As opposed to the previous example, where two gadgets were used (one to zero out the target register, and one to add a constant to the register), in this example one is sufficient. In this scenario, the desired constant is injected together with the addresses of gadgets. During execution, the gadget stored at `@Gadget1` simply loads the constant from memory into the appropriate register. This technique actually gives the attacker the advantage of being able to choose the constant freely, while with the other example, the attacker has to rely on a combination of arithmetic gadgets to create the desired value dynamically. However, depending on the technique used to inject the data into the program's memory space, it may not be possible to inject arbitrary values. Keep in mind that `pop` is equivalent to `mov r64, [rsp]`, i.e., loads 8 bytes in the target register. Therefore, if `rax` should contain `40h`, the attacker actually needs to inject `0000000000000040h`, which obviously contains null bytes, which may or may not be prohibited.

So far we have implicitly assumed that `rsp` points to the ROP gadgets injected by the attacker. However, this is not necessarily always the case. Think of, for example, a buffer overflow where the attacker has not enough space on the stack to inject the necessary gadgets, but is able to inject them on the heap. In this case, the attacker first has to point `rsp` to the memory location where the gadgets are located. To achieve this, a *stack pivot gadget* is used. This special kind of gadget ensures that `rsp` points indeed to the injected gadgets and must therefore be executed as the very first gadget. For example, if the attacker knows that the injected gadgets are located somewhere in memory and that `rsi` points to them, a useful stack pivot would be `xchg rsp, rsi ; ret`. The `xchg` instruction switches the values of `rsp` and `rsi`, so afterwards `rsp` points to the attacker-controlled gadgets. Then, the `ret` instruction works as usual, i.e., executes the next gadget.

Since ROP gadgets can be combined arbitrarily, ROP is a very versatile technique. In fact, ROP has been shown to be Turing complete on several platforms [90,135,144], therefore allowing arbitrarily complex computations. This means that any shellcode an attacker wants to execute in a regular code injection attack, she can also create using ROP. This effectively bypasses DEP and makes ROP extremely dangerous.

A main problem of ROP is, however, that the addresses of gadgets have to be known to the adversary beforehand, which is exactly what ASLR attempts to prevent: as we have shown in Chapter 2.3.2, ASLR ensures that binaries are always located at a different address for each execution. Therefore, if all modules are protected by ASLR, an attacker has to exploit one of ASLR's weaknesses we described in Chapter 2.3.2 to learn an address first.

2.4.2 Jump-Oriented Programming

Like ROP, JOP [18, 28, 107] also reuses short sequences of code, however, JOP gadgets end with an indirect jump, e.g., `jmp rax`. Therefore, JOP lacks a convenience the `ret` instruction at the end of ROP gadgets brings naturally: since `ret` moves `rsp`, it automatically points to the next gadget. To make JOP gadgets execute consecutively, JOP requires a *dispatcher gadget*; this special gadget moves the register that points to the gadgets forward, so it points to the next one. Listing 2.17 and Listing 2.18 show simple dispatcher candidates. This behaviour is similar to the one in the previous chapter about ROP, where we described how `rsp` can be considered another program counter. In JOP, any register can serve in this function, however, at the cost that the register has to be moved to the next gadget using a dispatcher gadget.

This implies that only gadgets whose `jmp` instruction targets the register pointing to the dispatcher gadget can be used. Therefore, having only one register point to the dispatcher gadget means that the number of JOP gadgets will likely be small, as only gadgets that end with a jump to this one register can be used. On the other hand, if there are several registers pointing to the dispatcher gadget, those registers must not be changed by any of the gadgets. Another difficulty arises from the limitation that the register used by the dispatcher to invoke the gadgets must also not be changed, meaning that JOP makes at least two registers unusable. Figure 2.20 summarizes the necessary steps of a JOP exploit.

JOP is even more cumbersome than ROP, due to the restriction described in this chapter. Currently, it has almost no relevance in real exploits, however, since many mitigations target ROP and `ret` instructions specifically, JOP might turn into a viable alternative.

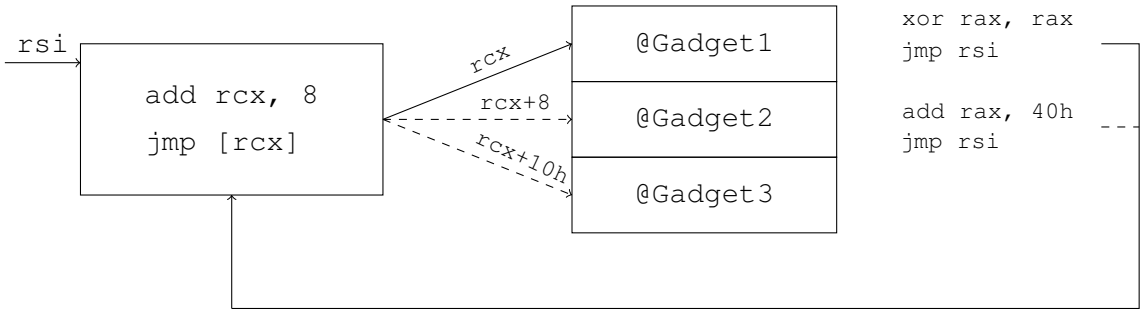


Figure 2.20: A schematic overview of JOP. The dispatcher gadget on the left, pointed to by `rsi`, increments the register pointing to the gadgets, in this case `rcx`, so it successively invokes all gadgets. The gadgets have to end with an indirect jump to the register pointing to the dispatcher gadget, i.e., `jmp rsi` in this example.

Listing 2.17: JOP dispatcher gadget

```
add     rax, 8
jmp     [rax]
```

Listing 2.18: JOP dispatcher gadget

```
add     rax, rdi
jmp     [rax + 10h]
```

2.4.3 Gadgets

Schacham [144] and Roemer et al. [135] describe different gadget classes, such as load, store, control flow, arithmetic, and logic. They focus, however, on showing that ROP is Turing complete, which is rarely required in practice. Indeed, special gadgets which are used in certain situations are more important in real exploits.

In this chapter we introduce these gadgets, and also show why the x86-64 architecture inadvertently supports code-reuse attacks better than some other architectures.

Special Gadgets

In some scenarios, specific gadgets are required. We have already introduces the *stack pivot* gadget, which is used when `rsp` does not point to the injected gadgets. Basically, any gadget that changes `rsp` is a stack pivot candidate. If `rsp` points somewhere close to the attacker-controlled buffer, a simple addition or subtraction to `rsp` might suffice. In other scenarios, where another register points to the buffer, loading the value of this register into `rsp` is necessary.

Another gadget that is sometimes necessary is the *write null* gadget, that writes a series of zeros to memory. This gadget held a bigger importance a few years ago, when x86 was predominant, where parameters were passed on the stack. Depending on the vulnerability,

an attacker might not be able to use null bytes in her payload, prohibiting injecting a parameter containing null directly. Usually, such a gadget first zeroes a register, e.g., using `xor rax, rax`. Then, the value of that register is written to a memory location pointed to by another register, e.g., `mov [rbx], rax`. A *write-what-where* gadget is the generalization of this. It is a simple primitive that allows writing arbitrary data at arbitrary memory locations, provided the the exploit developer can control these registers.

Some newer mitigation techniques, which we discuss in Chapter 3, use heuristics to detect abnormal program behaviour, such as long sequences of short basic blocks. Since attackers mainly rely on short gadgets that achieve a simple task, such as zeroing a register or writing a value to memory, such heuristics have a good chance of detecting them. Therefore, using special gadgets consisting of many instructions which do not use many registers to bypass such mitigations have been proposed by Davi et al. [49], Göktaş et al. [71], and Carlini and Wagner [26]. They dubbed such gadgets *long-NOP gadget*, *heuristic breaker*, or *long termination gadget*, respectively.

If an attacker wants to invoke a system call directly instead of going through a corresponding API, she needs a *syscall* gadget. On 64-bit Linux, this is a simple `syscall` instruction, with the number of the syscall stored in `rax`.

Unintended Instructions

Lastly, we look at an interesting property of the x86-64 architecture. Instructions are not aligned and of varying length (1 to 15 bytes), which allows for *unintended instructions*. An unintended instruction is an instruction that results from disassembling from a different offset than originally intended. Consider the following byte stream: `488D450F490F43C2482BD8`. When disassembled from the start, it results in the instruction sequence shown in Listing 2.19.

Listing 2.19: Disassembly of `488D450F490F43C2482BD8`

```
1 lea     rax, [rbp+0fh]          ; 48 8D 45 0F
2 cmovnb  rax, r10                ; 49 0F 43 C2
3 sub     rbx, rax                ; 48 2B D8
```

Due to the high density of the x86-64 instruction set, it is likely that disassembling from an arbitrary offset into this byte stream yields legal instructions. Therefore, unintended instructions are very common, as our example shows: continually skipping one byte from the start, results in the disassemblies shown in Listing 2.20 - Listing 2.25.

Listing 2.20: Disassembly of `8D450F490F43C2482BD8`

```
lea     eax, [rbp+0fh]          ; 8D 45 0F
cmovnb  rax, r10                ; 49 0F 43 C2
sub     rbx, rax                ; 48 2B D8
```

Listing 2.21: Disassembly of `450F490F43C2482BD8`

```
cmovns      r9d, [r15]          ; 45 0F 49 0F
ret         2b48h               ; 43 C2 48 2B
-           -                   ; D8
```

Listing 2.22: Disassembly of `0F490F43C2482BD8`

```
cmovns      ecx, [rdi]          ; 0F 49 0F
ret         2b48h               ; 43 C2 48 2B
-           -                   ; D8
```

Listing 2.23: Disassembly of `490F43C2482BD8`

```
cmovnb  rax, r10                ; 49 0F 43 C2
sub     rbx, rax                ; 48 2B D8
```

Listing 2.24: Disassembly of `0F43C2482BD8`

```
cmovnb  eax, edx                ; 0F 43 C2
sub     rbx, rax                ; 48 2B D8
```

Listing 2.25: Disassembly of `43C2482BD8`

```
ret         2b48h               ; 43 C2 48 2B
-           -                   ; D8
```

The instructions in Listing 2.21 and Listing 2.22 are especially interesting because, as they end with `ret`[20], they can be used for ROP exploits. This shows that, by jumping in the middle of an instruction, new and possibly useful gadgets can be generated. Neither hardware nor software are aware what the correct disassembly should look like, therefore this behaviour is not prohibited.

2.5 Summary

In this chapter we presented background information on which the remainder of this dissertation builds upon. Chapter 2.1 introduced some aspects of the x86-64 architecture, required to understand how low-level memory structures are organized and how function calls work. Chapter 2.2 elaborated on widely exploited bugs and presented techniques how they can be exploited. Chapter 2.3 presented common mitigation techniques, designed to prevent such exploits. Lastly, Chapter 2.4 introduced ROP, the currently most important exploitation technique, which requires knowledge of all topics presented previously in this chapter.

In the next chapter we discuss related work and revisit some of the concepts presented here to discuss them in more detail.

[20]The two bytes after the instruction are added to `rsp`, which is normally used by the compiler to clean up the stack from passed parameters. E.g., if, on Windows, a function took six parameters, the compiler will likely emit a `ret 10h` instruction.

Chapter 3

Related Work

In this chapter we discuss related work. It is divided into three sub chapters. Chapter 3.1 and Chapter 3.2 discuss mitigations against attacks and attacks against mitigations. While the focus is on current scenarios, i.e., mitigations against attacks used in the wild such as use-after-free, and attacks against real mitigations, such as ASLR, we also discuss attacks and mitigations which are of academic nature and are, at least currently, not used in real attacks. Lastly, Chapter 3.3 discusses work related to gadgets and automation.

3.1 Mitigations

Shadow Stack

The concept behind a shadow stack is to have an area of memory where backup copies of all return addresses are stored. Upon returning from a function, the return address stored on the stack is compared to the one on top of the shadow stack. During correct program execution they should be identical. Using a shadow stack battles ROP attacks, because an attacker would have to overwrite both the return address on the stack and the one on the shadow stack. However, typically, the shadow stack is protected, e.g., using guard pages or hidden in memory without any user-space accessible pointers to it. Shadow stacks can only protect return addresses, but provide no protection against JOP attacks or attacks targeting, e.g., vtables. Shadow stacks can be considered a fine-grained form of backwards-edge CFI.

Many different implementations exist [8, 34, 40, 47, 50, 74, 88, 110, 121, 130, 147], which differ in regards to performance, as a result of whether they're implemented as compiler extension, using binary rewriting, or dynamic binary instrumentation. Dang et al. [45] give an excellent overview and found the average overhead to be about 10%.

As oposed to software implementations, hardware implementations like StackGhost [65], HAFIX [47], or Intel's Control-flow Enforcement Technology (CET) [85] have almost no measurable performance impact. StackGhost is implemented for the SPARC architecture,

HAFIX for Intel Siskiyou Peak and SPARC, and CET will be part of Intel's future x86-64 processors. There is, however, no information regarding a release date at the time of writing.

Bounds Checking

Bounds checking fights the problem of memory corruption at its root and tries to prevent buffer overflows. It can be implemented as a compiler extension, where it works on source code, where information about buffers and pointers is readily available [2], or on compiled binaries [149], which uses debugging symbols or tools like Howard [148] to reconstruct buffer information from stripped binaries. Once buffers and potentially unsafe accesses have been identified, code is added to secure these accesses. Depending on whether the implementation is based on source code or on the binary level, as well as the granularity, i.e., the object-level or variable level, the overhead lies between 10% and 2 to 3x. PAriCheck [171] takes a slightly different approach and marks pointers that go out-of-bounds as the result of an arithmetic operation as invalid. It achieves this by assigning a label to each memory area inhabited by an object and, upon pointer arithmetic, checks whether the label at the beginning of the operation is the same as the label of the resulting pointer after the operation has finished. PAriCheck requires recompilation and exhibits a runtime overhead of about 49%.

Control-Flow Integrity

We introduced the general concept of CFI in Chapter 2.3. Here, we discuss the various implementations and give an overview in Table 3.1.

To make CFI practical in terms of performance and applicability, many CFI implementations are very coarse-grained [1, 85, 88, 175]. They classify indirect control-flow instructions and indirect control-flow targets into groups and enforce policies so indirect call-sites match the possible targets. For example, a simple policy might use just two categories, also referred to as labels, and enforce that all `ret` instructions must return to a call-preceded instruction and all indirect calls must target a function entry point. While this drastically reduces the number of available gadgets, many attacks, which we introduce in the next Chapter, have shown that they can be bypassed because they are overly permissive.

More fine-grained approaches [108, 125, 162, 164, 165, 174] increase the number of categories, taking even more leeway away from an attacker. A popular approach taken by many mitigations [85, 88, 125] to protect backward-edges, i.e., `ret` instructions, is using a shadow stack. Other tools [162, 165] use information about arity to create more groups, i.e., enforce that a `call` that prepares n arguments can only invoke a function that consumes at most n arguments. CCFIR [174] creates an own group for sensitive functions, which are not allowed to be targets of indirect branches. Despite these approaches, even fine-grained

CFI can sometimes be bypassed, as research in Chapter 3.2 shows. We deem a shadow stack essential to protect backward-edges, as policies enforcing call-precedence can be bypassed fully automatically using PSHAPE, as we show in Chapter 5. Another large drawback of current CFI implementations is that they do not protect JIT-compiled code. Burow et al. [24] offer a detailed comparison of many CFI approaches.

Heuristic-based Code-Reuse Attack Detectors

Heuristic-based solutions monitor programs at runtime and detect "unusual" behaviour that can be linked to code-reuse attacks. The difficulty of these approaches is defining what unusual behaviour means in terms of code-reuse attacks. Most implementations [29, 33, 64, 120], including our own called ROPocop, rely on two distinct features current attacks exhibit, namely a large number of consecutive indirect control-flow transfers and short basic-blocks. Table 3.2 gives an overview over popular approaches.

To increase performance, some implementations [33, 120] use a debugging / profiling feature known as *last branch record* (LBR) many modern CPUs offer. LBR stores the source and destination address of the previous n branches, n currently being between 4 and 32, depending on the CPU architecture [84] (Vol 3B, 17.4.8.). In addition, they also do not constantly monitor the parameters mentioned above, but only at certain key events, such as when syscalls are performed. While solutions using dynamic binary instrumentation [29,64] exhibit an overhead which is between 2 to 5x, implementations using LBR, which forego constant monitoring at the cost of security, reduce the overhead to about 2%.

HDROP [176] uses a slightly different approach, and relies on a CPU's branch prediction, or, in the event of ROP attacks, a large number of mispredicted return instructions. Since these can be monitored using hardware performance counters (HPC), this approach is relatively fast, at an overhead of 19%. It does, however, require recompilation of the program because it inserts checking points, where it reads and interprets the HPC values.

These solutions can only provide probabilistic security and work best against an unaware attacker. If the attacker knows that a heuristic is monitoring program execution in the background, she can try to use longer and more complex gadgets in her exploit in order to avoid detection. Doing this manually is difficult, however, but we show that these heuristics can be bypassed automatically by PSHAPE in Chapter 5.

Heap Protection / Use-After-Free Mitigation

Mitigations in this category aim at hindering exploitation of use-after-free bugs. We further divide them into two broad groups. Approaches in the first group, namely FreeSentry [170]

Table 3.1: Overview of CFI implementations

Tool	Implementation	Details	Overhead
CFI [1]	Binary rewriting	Uses Vulcan [57] for CFG extraction and binary rewriting. Return must target call-preceded instruction. Original proposal.	16%
BR [88]	Binary rewriting	Uses binary annotations based on symbols. Protects backward-edges with a shadow stack. Requires hardware changes.	2%*
O-CFI [108]	Binary rewriting	Transforms branch checking problem to bounds checking. Combines coarse-grained CFI and fine-grained ASLR.	4.7%†
CFI for COTS [175]	Binary rewriting	Return must target call-preceded instruction.	6.4%
CCFIR [174]	Binary rewriting	Prevents invocation of sensitive functions through indirect control-flow transfers. Return must target call-preceded instruction.	3.6%
Lockdown [125]	DBI	Uses libdetox [126] for instrumentation. Protects backward-edges with a shadow stack.	32.5%
TypeArmor [165]	DBI	Uses Dyninst [13] for instrumentation. Targets specifically COOP [138] attacks. Arity of call-site and target must match. Protects only forward-edges.	3%
CET [85]	Compiler extension	Hardware shadow stack and new instructions for CFI enforcement. Part of Intel's future CPU generations.	n/A‡
IFCC [162]	Compiler extension	Implemented in LLVM. Forces all indirect calls and jumps to go through a table containing legal targets. Arity of call-site and target must match. Protects only forward-edges.	4%
CCFI [164]	Compiler extension	Adds message authentication codes (MACs) to control-flow elements. Dynamically classifies pointers. More precise than static approaches.	10%

*Based on simulations with emulated hardware.
†Without Intel MPX. Estimated overhead using MPX is 4.17%.
‡Not available at the time of writing.

Table 3.2: Overview of heuristic-based mitigations

Tool	Monitored Parameters	Monitoring Interval	Implementation	Overhead
ROPocop [64]	Basic-block length and # of consecutive indirect branches	Constantly	PIN [98]	2.4x
DROP [29]	Basic-block length and # of consecutive indirect branches	Constantly	Valgrind [111]	5.3x
kBouncer [120]	Basic-block length and # of consecutive indirect branches	Before syscalls	Detours [82] and LBR	1%
ROPecker [33]	Basic-block length and # of consecutive indirect branches	Before syscalls	Linux kernel module and LBR	2.6%
HDROP [176]	Mispredicted returns and # of returns	Configurable	gcc module and HPC	19%

and DANGNULL [92] trace pointers at runtime and nullify all pointers to a freed object. This turns a use-after-free bug into a null-dereference. They require recompilation and incur overheads of about 42% (FreeSentry) and 80% (DANGNULL). The latter also reports large memory overheads of up to 5x. The former does not discuss memory overheads. Therefore, these approaches may not work well for applications using many objects, such as browsers.

The second group contains approaches which change the way memory allocation and deallocation works [12, 114, 169]. An *infinite heap*, as proposed by Berger and Zorn [12], spaces out objects "infinitely" far from each other. Of course this is not possible in practice, as virtual addresses are finite, which is why only approximations of such systems exist. This requires the heap to be M times larger than would be required. It allows applications to continue execution during an accidental buffer overflow. To increase security against attacks, DieHarder [114] improves the original design by increasing the entropy of the algorithm that selects where in memory an object is stored.

Microsoft uses Memory Protector [169] in Internet Explorer. It consists of an *isolated heap* [160] and *delay free* [161]. The isolated heap allocates objects which are often used in conjunction in use-after-free attacks on different heaps, e.g., strings and `IE element` objects. A similar technique has been added to the Adobe Flash allocator [22]. Delay free does not deallocate memory immediately, but instead adds a freed object to an array. The objects in this array are only freed under certain conditions, making it more difficult for an attacker to reliably force freeing memory. Microsoft introduced *safe unlinking* and *reference*

count hardening [102] in Windows to further increase heap security against certain classes of attacks.

Virtual Function Table Protection

Corruption of vtables is currently a primary attack vector in many exploits. Therefore, many researchers proposed solutions to prevent such attacks [20, 67, 75, 86, 162, 172, 173]. vtable corruption and vtable injection attacks are easily thwarted by enforcing that all vtables are in a memory region that does not grant write permissions. However, this still leaves the option of reusing existing vtables or data that can be interpreted as vtable, similarly to code-reuse attacks. Table 3.3 gives an overview over the various defences against such attacks. These defences can be considered forward-edge CFI for C++.

To prevent vtable reuse attacks, there are different approaches. Some, based on source code, build a class hierarchy and insert checks to only allow invocations of functions in vtables matching the base class or its derived classes [75, 86, 162, 173].

Others [67, 129, 172] add checks to dispatchers to decrease the chance that random data can be interpreted as vtable, increasing security probabilistically. For example by querying a random vtable entry and checking that it points to non-writeable memory [67]. This implies that data close to the fake vtable entry the attacker wants to use also needs to be a valid code pointer. Another option is inserting an ID before legitimate vtables [172]. In this case, data before the fake vtable entry the attacker wants to use has to match the ID. vfGuard [129] takes yet another approach: it discovers vtables using signatures and enforces a policy that specifies that an indirect call invoking the nth function can only target vtables which contain at least n functions.

Solutions generally exhibit a very low overhead in the low, single-digit area, which was reduced even further by Bounov et al. [20] who implement a new way of storing and checking vtables, reducing overheads to just 1% for compiler-based approaches.

Type Confusion Mitigation

CAVER [93] is implemented as LLVM extension. It stores meta-data that contains for an object of type T, which types it can be casted to. A runtime library verifies that casts adhere to the stored data. The authors instrumented Chromium and Firefox and their evaluation shows that CAVER adds an overhead of about 33%. TypeSan [76] improves upon this design and has better coverage at lower overhead of approximately 22%.

Table 3.3: Overview of vtable protection schemes

Tool	Prevention of vtable reuse	Implementation	Overhead
T-VIP [67]	Probabilistic	Binary rewriting	2.2%
VTint [172]	Probabilistic	Binary rewriting	2%
vfGuard [129]	Probabilistic	DBI	18%
VTrust [173]	Class-tree hierarchy	Compiler extension	2%
SafeDispatch [86]	Class-tree hierarchy	Compiler extension	2.1%
VTV [162]	Class-tree hierarchy	Compiler extension	8.4%
ShrinkWrap [75]	Class-tree hierarchy	VTV [162] Extension	5%

Address Space Randomization

Approaches in this chapter aim to improve current ASLR, which only randomizes on the binary level, i.e., only the start address of a binary is randomized. This enables an attack vector in which the attacker can leak a single address inside that binary, e.g., by reading a return address from the stack. Since only the start address of the binary is random, all offsets within the binary are constant, giving the attacker full knowledge in regards to the addresses of all gadgets. Table 3.4 gives an overview over the various proposals.

Early approaches suggested up until around 2013 improve shortcomings of ASLR at that time, i.e., randomize the main binary [14, 51, 77, 89, 119, 168], and/or randomize on a lower level, e.g., on a function [89], basic block [51, 168], or even instruction level [77, 119]. They work on binaries and do not require source code, allowing for effective randomization. However, with the introduction of JIT ROP [150] and a growing number of information leak vulnerabilities, these solutions became moot, as using these techniques, an attacker can disclose memory at runtime.

Many new approaches aim at preventing the leakage of code pointers [7, 21, 42, 43, 97]. They use a variety of techniques to achieve this: i) complete separation of code and data. ii) enforcement of execute-only memory. iii) hiding forward pointers by using trampolines redirecting to execute-only memory. iv) encrypting return addresses. On top of that, all approaches use fine-grained randomization. Most of them, however, do not protect JIT-compiled code, enabling a new attack vector [5]. Their overheads are between 1 to 6%.

Other mitigations take different approaches: Fine-Grained ASR [69] and TASR [16] use runtime rerandomization. Fine-Grained ASR rerandomizes in fixed intervals, leading to varying overheads between 50% when randomizing once every second to 10% when randomizing every ten seconds. TASR, on the other hand, rerandomizes only before program input and after program output. The argument behind this is that any program output could potentially be an info leak, therefore the memory layout is changed, rendering the disclosed

information stale. To further increase security, rerandomization is also applied before processing any input, as input can potentially be an exploit. The authors only evaluated TASR against SPEC CPU2006, which has barely any input or output, therefore low overheads of 2.1% are no surprise. Unfortunately, they did not apply TASR to real programs.

Lastly, Isomeron [48] loads the original program and a diversified copy in parallel. At runtime, execution switches between the two versions. A switch can appear before any control-flow transfer. Isomeron does not try to hide any information from the attacker but instead provides probabilistic security: each gadget the attacker uses reduces the chance of success. If she needs just one gadget, the chance of guessing the right binary is 50%. For two gadgets the chance of guessing correctly decreases to 25%. Considering many contemporary ROP chains use around ten gadgets, the chance of success decreases to less than 0.1%. In Chapter 5 we present a single gadget that loads all registers used for passing parameters, which greatly increases the chance of success. PSHAPE is able to find such gadgets fully-automatically.

Execute-Only Memory

Fine-grained ASLR improves over ASLR by removing constant offsets even within a binary. In such scenarios, an attacker repeatedly exploits an info leak to read pages containing code, extract gadgets, and construct a ROP chain at runtime. We discuss this technique called JIT ROP [150] in more detail in Chapter 3.2.

Two systems aim to mitigate this threat [6,159]. They both identify the root problem that code that is executable is automatically also readable. This allows an attacker to read code like data, construct a ROP chain when enough code has been read, and then execute it.

Backes et al. [6] introduce the notion of *execute-no-read* (XnR), which allows code to be executed, but not read as data. It intercepts all read accesses to code or data and determines the kind of access, i.e., reading code to execute it (known as instruction fetch operation), reading data, or reading code as data. It is implemented in the Linux kernel and exhibits an overhead of about 3%.

Heisenbyte [159] takes a slightly different approach they call *destructive code reads*. Here, code may be read as data, but if this is detected, the code is garbled after the reading operation. It works on COTS binaries and introduces an overhead of about 17%.

Miscellaneous

Microsoft EMET [106] includes several ROP detection mechanisms, e.g., caller checks, which make sure that critical functions are invoked via `call` and not `ret`, or a routine that detects

Table 3.4: Overview of ASLR improvements

Tool	Implementation	Details	JIT code	Overhead
Oymoron [7]	Binary rewriting	Hides code pointers located in code pages using trampolines. Can be bypassed [48].	No	2.7%
Readactor [42]	LLVM	Separates code and data. Enforces XoM using a thin hypervisor. Hides code pointers using trampolines.	Yes	6.4%
Readactor++ [43]	LLVM	Similar to Readactor. Adds further randomization to prevent reuse of whole functions.	Yes	1.1%
LR^2 [21]	LLVM	Implements known techniques on ARM to introduce leakage-resilient randomization to embedded and mobile devices.	No	6.6%
ASLR-GUARD [97]	GCC	Uses a secure stack for sensitive stack data (e.g., return addresses). Hides code pointers using trampolines.	No	1%
TASR [16]	GCC	Rerandomizes after program input and output.	No	2.1%*
Fine-Grained ASR [69]	LLVM	Uses live-rerandomization in certain intervals.	No	50%†
Isomeron [48]	DBI	Executes the original program and a diversified copy in parallel. Switches between them at runtime.	No	2.2x

*SPEC benchmarks have little input and output, therefore real numbers regarding performance do not exist.
†Measured using a microkernel, randomizing once every second. Randomizing every ten seconds leads to an overhead of about 10%.

stack pivoting. EMET works directly on binaries and is very easy to use, however, many versions of EMET have been bypassed rather quickly after their release [115, 131].

Code-pointer Integrity (CPI) [91] splits memory into a regular and a safe region. It uses static analysis during compilation to identify which code pointers need to be protected and places these on a SafeStack which is hidden in the safe region. It introduces an overhead of about 5%. Evans et al. showed how it can be bypassed [58], outlining that the main issue is that the safe area is not well enough hidden in memory, and can be found. Their attack, however, is either noisy, causing many crashes, or slow, taking several days to succeed.

Li et al. [94] aim at preventing ROP by creating a compiler extension for LLVM which emits code that does not use `ret` instructions and instead uses direct `jmp` instructions. Return addresses are stored in read-only memory. It allows an attacker to return to an arbitrary return address and does not protect from other forms of code-reuse attacks. It introduces an overhead of about 11%.

G-Free [117] enforces that all functions must be entered through their header, aligns instructions to get rid of unintended instructions, and encrypts/decrypts return addresses in the function prologue/epilogue, respectively. It is implemented as a compiler extension and introduces an overhead of 3.1%.

RUNTIMEASLR [96] aims to prevent a very specific kind of attack, namely, *clone-probing* attacks (e.g., BROP [17] which we introduce in Chapter 3.2). This attack is used to break ASLR and relies on the assumption that services like a daemon process fork a new process for every connection. These forked processes all inherit the original memory layout of the main process, allowing brute force attacks. RUNTIMEASLR re-randomizes forked processes, mitigating such attacks without a measurable runtime overhead, but at the cost of longer startup time.

Graffiti [44] is a tool that detects heap spraying. It is based on a hypervisor which analyses memory at runtime. Graffiti is OS-independent and allocator-agnostic which allows deployment on a wide variety of applications. It detects certain patterns, such as those heap sprays or data sprays exhibit. Similarly, NOZZLE [133] also detects heap sprays using certain heuristics, e.g., looking for NOP-sleds, and interpreting object data as code and performing static analysis to detect malicious intent. HexPADS [124] also uses patterns to detect a variety of attacks, such as side-channel or covert-channel attacks. HexPADS uses hardware performance counters to measure cache behaviour, which can be an indicators for certain attacks.

Stancill et al. [156] propose a framework which analyses documents and detects ROP payloads. It opens a document in the native application, takes a memory snapshot, scans for gadget chains, and, if chains are found, profiles their overall behaviour. This technique, which aims at detecting attacks, is very different from approaches that aim at preventing

attacks. It is much simpler to deploy and has no compatibility issues. However, since it uses many heuristics, it can potentially be bypassed by an aware attacker.

StackArmor [31] protects against stack-based attacks, working on binaries without the need for source code or debugging symbols. Using static analysis, it discovers buffers using reverse-engineering techniques and decides which buffers are potentially vulnerable. In the next step, it rewrites the binary to allocate vulnerable buffers in new, isolated, and randomized stack frames and updates all references accordingly. Without debugging symbols, StackArmor relies on many heuristics to safely move buffers to other memory regions, which may lead to unprotected buffers. StackArmor introduces an overhead of about 28%.

3.2 Attacks

ASLR

Serna [143] describes info leaks, a general term for a wide range of vulnerabilities that allow an attacker to infer information about the memory layout. One popular approach is leveraging a heap overflow to overwrite the first bytes of a Javascript string. These bytes store the length of the string, and by overwriting them with `0xffffffff` an attacker can access arbitrary memory locations using Javascript read primitives.

Leakless [60] executes arbitrary functions in any loaded library in the presence of ASLR. It abuses the ELF dynamic loader's lazy symbol resolving functionality to resolve a function name the attacker injected. Leakless achieves this by crafting data structures used to resolve functions and overwriting pointers which point to them. One large drawback is that it does not work if the main binary is compiled as position-independent executable (PIE). Furthermore, depending on whether the binary is compiled with partial or full RELRO, it needs specific gadgets. PSHAPE can help automate this attack by finding these gadgets.

Cross-VM ASL INtrospection (CAIN) [9] targets processes running in a virtual machine (VM). It exploits a memory page deduplication side-channel: when the same page exists in two guests, it is merged to save resources. Subsequent writes to such a page are slower, hence allow an attacker to know when a page has been merged. To exploit this, the attacker crafts pages that must exist in the victim VM, guessing their base-addresses. Once the correct base-address has been found, it will be merged, which the attacker can measure by writing to it again. HexPADS [124] can detect and slow down this type of attack.

JIT ROP [150] bypasses fine-grained ASLR by constructing the payload at runtime using a scripting environment, e.g., Javascript, commonly found in browsers, or Actionscript, found in many Adobe products. Based on just one code-pointer (leaked through arbitrary means, e.g., a regular info leak) and a *DiscloseByte* primitive, which takes an address and discloses its content, JIT ROP automatically harvests code pages and discovers gadgets. The

underlying concept is that the initial code-pointer discloses a whole page (4 KiB). This page can be read using the *DiscloseByte* primitive. In these 4 KiB of code, there are likely more code-pointers which can easily be found by looking for `call` and `jmp` instructions and which point to new pages. These steps can be repeated until a large part of memory has been mapped. This attack renders fine-grained ASLR approaches useless. Rerandomization approaches work well, however, as real JIT ROP attacks take up to 23 seconds.

Seibert et al. [141] implement remote side channels and found that simply executing a program leaks information about the code. By modifying certain data in memory and employing fault analysis and timing information, knowledge about the program state can be obtained. Modifying data requires a memory corruption vulnerability that allows the attacker to overwrite arbitrary data, e.g., variables on the stack, return addresses, or code pointers. Depending on what she can overwrite, she can infer information about the memory layout. For example, by overwriting a code pointer with an arbitrary address and timing how long it takes to execute, the attacker may be able to identify which function is executing.

Blind ROP (BROP) [17] introduces an interesting ASLR bypass, but only works as long as the target application contains a stack-based vulnerability and restarts after a crash without re-randomizing its memory layout. Several nginx or apache vulnerabilities fit this profile. To bypass ASLR in such a scenario, simple brute-forcing could be used: the attacker overwrites the return address with 0 and increases the value till the program continues without crashing. When this happens, the correct return address has been discovered, i.e., a code pointer inside the main library has been found and, due to the constant offsets, addresses of all gadgets are now known. The basic idea of BROP's *stack reading* technique is to overwrite the target data byte by byte. The last byte can be one of 256 values, i.e., it takes, on average 128 attempts to guess it. Then, the next byte can be guessed, which again takes on average 128 bytes. Therefore, to guess the return address or the stack canary on a 64 bit system, about 1,024 attempts are necessary.

CFI and Heuristic-based Mitigations

In 2014 and 2015 many attacks targeting various CFI implementations and heuristic-based mitigations, e.g., kBouncer [120], ROPecker [33], or CFI for COTS [175] have been published. As these mitigations place tight restrictions on indirect control-flow transfers, hence also gadgets, those attacks often incorporate gadgets that would rarely be used in real attacks. E.g., Carlini and Wagner [26], Davi et al. [49], and Göktaş et al. [71], and Schuster et al. [139] discovered that long gadgets with few side effects are suitable for breaking heuristics-based mitigations. Such gadgets should consist of at least 20 instructions, preserve as many registers as possible, have few side-effects, and easily fulfillable preconditions. PSHAPE makes

Table 3.5: Overview of ASLR attacks

Attack	Requirements	Impact
Info Leaks [143]	Info leak vulnerability	Vulnerability-dependent*
BROP [17]	Stack-based overflow, forking application	Leaks arbitrary stack value (e.g., canary, return address)
CAIN [9]	Process inside a VM	Leaks library base address
Leakless [60]	Non-PIE ELF binary, certain gadgets	Invokes arbitrary function in any loaded library
JIT ROP [150]	A code-pointer, scripting environment	Leaks large parts of memory layout
Seibert et al. [141]	Memory corruption vulnerability	Vulnerability-dependent

*May range from allowing the attacker to read just one byte at a fixed address to reading arbitrary addresses

finding such gadgets very easy, as we show in Chapter 5. Another kind of gadget commonly used in these attacks is an LBR-flushing gadget [26, 139]. LBR is used by kBouncer [120] and ROPecker [33] to store the last n addresses of taken branches. When certain, critical APIs are invoked, the LBR is inspected and, depending on whether the control-flow appears legitimate or not, an exception is raised. LBR-flushing gadgets are gadgets that naturally contain many indirect branches, present in the regular control flow, e.g., functions that call lots of sub-functions. By using such a gadget, the LBR is filled with legitimate addresses and there is no trace of irregular control flow, i.e., ROP, in the LBR.

Counterfeit Object-Oriented Programming (COOP) [43, 138] is a technique to exploit C++ programs which reuses whole virtual functions. Useful computation is achieved solely through the side effects of whole functions. In a COOP attack, the attacker injects fake objects and repeatedly invokes virtual functions on them. This is achieved by using a main-loop gadget (ML-G) which iterates over an array of objects and invokes a virtual function on each of them. The authors show that COOP can be used to bypass many variants of CFI and vtable protection.

Stackdefiler [37] presents implementation errors in real CFI implementations, namely IFCC and VTV, which allow bypassing them. The biggest threat is arguably that callee-saved registers, which often hold data used for CFI checks, are spilled on the stack, hence an attacker may be able to overwrite them. According to the authors, this holds true for about 26% of all functions in Chromium. They also created a fix, which results in an increase of the overhead of about 0.5%.

Carlini et al. [25] discuss the effectiveness of theoretic CFI implementations against ROP attacks. Instead of evaluating real implementation, which usually take shortcuts to increase performance, they base their assessment on a perfect, fully-precise implementation. They

find that a shadow stack is essential for a strong defence, but even with a shadow stack, under certain circumstances attackers may be able to divert program flow and achieve malicious computations. Similarly, Control Jujutsu [59] shows that even fine-grained CFI solutions can be exploited. Over-approximation, which is necessary in order to avoid false positives, in conjunction with certain coding practices introduces edges an attacker can exploit. More precisely, the authors created exploits for apache and nginx, using gadgets that adhere to the CFG created by the state-of-the art DSA algorithm. They discuss shortcomings of the analysis and elaborate upon possible ways to improve it, while remaining practical.

Information Hiding

Gawlik et al. [68] introduce a crash-resistant memory scanning technique for web browsers. They abuse legitimate exception handling functionality to survive access to unmapped memory. In this case the exception handler, if installed, handles the error and may continue the program. Alternatively, syscalls may be abused. In this case, the return status code indicates whether a given address is mapped or not. Next, with the ability to probe memory without crashing, they create a *memory oracle* in Javascript, which, given an address, either returns a byte of data or an error. Together, this creates a powerful primitive that allows reading arbitrary memory locations without causing program crashes.

Göktaş et al. [72] discuss the security of information hiding. Many mitigations we introduced in Chapter 3 rely on data structures "hidden" in memory which are only accessed through segmentation registers or registers which are never spilled to memory, preventing an attacker from obtaining their location. They developed a technique called *thread spraying*, which they demonstrate on Firefox by finding the SafeStack CPI [91] uses. First, they force the program to allocate a large number of threads, therefore also a large number of SafeStacks. Next, for every thread, they create a large amount of repetitive but thread-unique data that will be deemed "safe" by CPI, hence placed on the SafeStack. This allows them to identify each individual SafeStack. Next, using a simple brute-force search and the previously introduced non-crashing memory scanning primitive [68], they scan the memory to discover a SafeStack. The authors report that their attack takes about 46 seconds. A drawback of this attack is, however, that spawning such a large number of threads is noisy and can potentially be very resource-intensive.

Oikonomopoulos et al. [116] take a different approach and infer information about the memory layout by recovering information about unmapped memory holes between mapped memory. They require the attacker to be able to read from and write to arbitrary memory, cause memory allocations of arbitrary size, and the ability to know whether an allocation was successful or not. They then repeatedly allocate memory and see if the allocation succeeds. Thereby, they are able to infer information about the size of holes in the memory

layout. By continually discovering holes, they also know where no hole is and can probe it using previously introduced techniques.

Miscellaneous

Athanasakis et al. [5] present an attack where the adversary targets code generated by JIT compilers. Such compilers, which generate native code from, e.g., Javascript, are omnipresent and used, for example, in browsers. This dynamically generated native code is often not protected by static approaches such as CFI, and allows an attacker access to new gadgets. In a scenario where the attacker sets up a malicious website and attempts to trick people into visiting it, she has full control over the Javascript code, hence can introduce arbitrary ROP gadgets. Modern JIT compilers attempt to make this process difficult, for example by obfuscating code or adding `nop` instructions. However, the authors show that these mitigations can be bypassed and that protection mechanisms, such as CFI, need to be applied holistically. Maisuradze et al. [99] build upon this attack and show that it can be used to bypass various mitigations that enforce execute-only memory, even when they protect JIT compiled code. They show that, since the attacker has the ability to create Javascript code that will be compiled to predictable native code, she does not need to read the code in the first place.

A similar attack by Snow et al. [151] against destructive code reads [159] also abuses JIT compilers. They force the JIT compiler to create two identical code sections of the same Javascript input. Then, they can use one copy to find gadgets and, after discovering the base address of the other copy, use all gadgets from the ungarbled code section. Next, they present a technique to force a browser to unload and reload a library, allowing an attacker to also first find gadgets, force an unload of the garbled library, force a reload of the same library, and, after discovering its base address, use all gadgets. They furthermore present an interesting technique they call *implicit reads*, which allows them to infer information about the code following already read code. For example, consider an ASLR implementation randomizing basic-block order of functions. An attacker can compute all possible orders of basic blocks for any given function. She can then read all basic blocks one after the other except for the last one, and she will know the contents of it.

In this dissertation we focus on attacks that hijack the control-flow. However, corrupting non-control data such as the user identity data, configuration data, or decision-making data can be a viable approach, as Chen et al. [30] show. Think of, for example, a computation at time t which results in a user's privilege level. The result is spilled to memory, and later at time t' used to grant the user certain rights. If an attacker is able to overwrite this value in memory between t and t', she can elevate her rights without corrupting any control-flow data. Chen et al. show that such attacks are real and demonstrated them on a variety

of servers, e.g., WU-FTPD, Null HTTPD, NetKit Telnetd, GHTTPD, and two SSH server implementations.

3.3 Gadgets and ROP Chains

ROPMEMU [73] helps analysing ROP chains. It works on a memory snapshot and uses emulation to create a CFG. Based on this CFG, regular reverse-engineering techniques can be applied to reconstruct a ROP chain's behaviour. This is an interesting and, so far, overlooked field of research. Especially with the automation of ROP chains and the ability to use long and complex gadgets, manual analysis of such chains will get increasingly difficult. This makes tools like ROPMEMU very important.

Q [140] takes an existing exploit which does not bypass DEP or ASLR, and attempts to harden it, i.e., rewrite it so it bypasses these mitigation techniques. To bypass ASLR it relies on unrandomized code sections and then uses gadgets from those sections to construct a ROP payload to bypass DEP. The payload is written by the attacker using QooL, Q's own exploit language. In their evaluation, the authors show how Q hardens nine simple stack buffer overflow exploits for Windows and Linux, with a payload that invokes a linked function or `system/WinExec`. Q cannot handle gadgets containing pointer dereferences, which PSHAPE not only handles, but also ensures they are safe to use.

ROPC [118] is based on Q, but publicly available. Its main feature, however, is not exploit hardening, but a gadget compiler, which takes an input binary and a program written in their own ROP language called ROPL. Then, ROPC creates this program using only gadgets from the input binary. Unfortunately, only a proof of concept prototype, dating back to June 2013, is available, which only works on a small synthetic example, but not on real binaries. The author also explicitly states that ROPC is a proof of concept and not practical, i.e., the opposite of PSHAPE.

nrop [167] helps analysts by finding semantically equivalent gadgets. For a given instruction and input binary, nrop outputs gadgets which are semantically equivalent to the input instruction. PSHAPE could be used for a similar purpose, as semantically equivalent gadgets have the same summary, i.e., pre- and postconditions.

Dullien et al. [56] create a framework for gadget discovery which works independently of the processor architecture. It finds gadgets on ARM, SPARC, MIPS, and x86-64 using an IR called Reverse Engineering Intermediate Language (REIL), which the authors developed. An input binary is first converted to REIL, where instructions that move the program counter are identified. Then, an analysis finds all paths leading to these instructions, resulting in a set of gadgets. All gadgets are analysed in terms of their functionality and then assigned to a category, based on which a Turing-complete set of gadgets is created.

Homescu et al. [78] present a hand-picked set of gadgets for the x86 architecture, where the size of the gadgets is limited to a maximum of three bytes. Their set of gadgets consists of 17 primitives, which is Turing-complete, allowing them arbitrary computations. In a realistic attack scenario, where Turing-completeness is not necessary, they built an exploit using a reduced set of gadgets, that contained only eight primitives.

There are many tools available that assist exploit developers to find and sort gadgets [38, 53, 109, 136, 137, 154], but none of them take into account the quality of gadgets. Some of these tools also attempt to automatically build a ROP exploit for one predefined scenario (e.g., ROPgadget [136], Mona.py [38], or ropper [137]), however, from our experience they are not very sophisticated and often fail, even if the necessary gadgets are available. We compare PSHAPE to these tools and discuss issues in Chapter 4.3.5.

ROPER [157] is a new project currently under development, that uses a very interesting and novel approach to ROP chain generation, a genetic component. It starts out with creating chains by randomly combining gadgets. Next, it selects four of these chains and assesses their fitness by executing them. The two least fit chains are discarded, the two fittest chains are *mated* and their children are added to the pool of randomly created chains. This process is repeated until one or several chains meet the user-specified machine-state. This is an interesting take on ROP chain generation, however, like PSHAPE, it will have to overcome the issue of state explosion. Randomly combining gadgets has to be controlled to some extent because otherwise the number of chains is infinite. This control mechanism, however, needs to be intelligent enough to not prevent the creation of possibly useful chains. Furthermore, this approach uses emulation and will therefore have to emulate the execution of a large amount of possibly complex chains. Assuming only 1,000 gadgets exist, the maximum length of the generated chain is limited to ten gadgets, and every gadget may only be used once in a chain, this results in almost one nonillion[1] possible chains[2]. This is only in the first step, before chains are mated.

[1] 1,000,000,000,000,000,000,000,000,000,000,000

[2] Computed as the total number of possible k-permutations of n objects: $P_{n,k} = \frac{n!}{(n-k)!}$

Chapter 4

Gadget Chaining

In Chapter 2.4.1 we introduced the basics of ROP. Now, in Chapter 4.1, we look beyond ROP's theoretical capabilities and elaborate on how it is used in practice and why it is currently the most important exploitation technique. Then, in Chapter 4.2, we present ROPocop, a tool that detects ROP attacks by monitoring program execution and searching for patterns typical for ROP. It detects ongoing attacks by inspecting the average length of basic blocks during execution and counting the frequency of indirect branches. Configurable thresholds allow tailoring its detection algorithms to individual programs. We also use ROPocop as an exemplary restriction the environment places on ROP development. Unaware attackers, i.e., if the attacker does not know that ROPocop is deployed, have a very high chance of being detected, but even an aware attacker will be forced to rethink her approach to exploit development in order to bypass ROPocop.

In Chapter 4.3 we propose a technique to automate ROP chain generation. Creating ROP chains manually is cumbersome, even without any restrictions. With defences, such as ROPocop, the task is even more difficult, as the developer has to utilize longer and more complex gadgets. Longer gadgets usually contain register aliasing, involve memory read and write operations, and access several registers. Manually determining all effects of gadget is time consuming and error prone. We propose PSHAPE, a tool that creates semantic *gadget summaries*, allowing a developer to immediately understand the effects of a gadget. Moreover, PSHAPE also chains gadgets together fully automatically, making sure that all dereferenced registers are user-controlled before being dereferenced. This allows an analyst to invoke arbitrary functions and ensures that the gadget chain does not crash due to accesses of unmapped memory. One large problem of ROP automation is state explosion. Randomly combining gadgets quickly leads to billions of combinations, even on very small sets of gadgets. We propose *smart permutations*, where we use only the gadgets which have the highest chance of being viable. This leaves us with only thousands of combinations,

which can be analysed quickly. In this chapter, we treat the algorithm that selects the most promising gadgets as a black box, and present details in the following chapter.

In Chapter 4.4, we discuss how gadget quality can be measured. We define a set of four metrics that grade individual gadgets and sets of gadgets and implement them in a tool called GaLity. Our algorithms unite several aspects of exploit building, look for gadgets required in real-world exploits, and calculate overall gadget quality. The results are used by PSHAPE to select the most well-suited gadgets.

4.1 ROP in Practice

ROP is the predominant form of code-reuse attacks, which is why we focus on it in this dissertation. As we have stated in Chapter 2.4.1, ROP is often Turing complete, and therefore allows arbitrary computations. However, writing shellcode using only ROP is a cumbersome task and such a payload will likely be very large. In reality, payloads are usually two-staged: Stage 2 runs regular shellcode, i.e., is similar to simple exploits that are able to inject and execute arbitrary data. Stage 1 uses ROP to invoke an OS function that changes the permissions of the memory page(s), where Stage 2 is stored, allowing it to run as if DEP were not present. Stage 1 ends with a control-flow transfer to Stage 2. Figure 4.1 shows this visually. Depending on the architecture, calling convention, environment, and availability of gadgets, Stage 1 can be very short (less than five gadgets) or very long, without an upper limit. The three biggest influences, apart from the availability of useful gadgets, are:

- Whether arguments are passed in registers or on the stack. This is dictated by the architecture and calling convention.

- Whether certain characters have to be avoided in user input. For example, if a `strcpy` causes an exploitable vulnerability, null bytes have to be avoided, because the function stops copying once a null byte is found. Another example for bad characters would be any non-printable ASCII characters when the input is supposed to be the path to a file.

- Whether the arguments are static and known in advance. For example, in a typical attack on Windows, an attacker might want to call `VirtualProtect` to change the protection level of some pages. Its parameters are the start address `lpAddress`, the size `dwSize`, the new protection level of the affected pages `flNewProtect`, and an address where the old protection level will be stored `lpflOldProtect`. If no randomization techniques such as ASLR are present, all these parameters are known beforehand, because `dwSize` and `flNewProtect` are constant and known to the at-

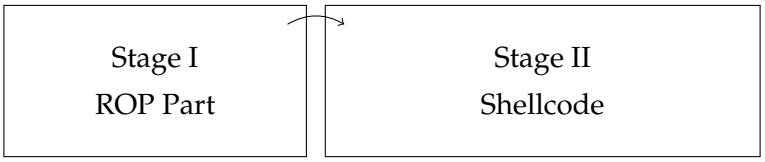


Figure 4.1: Structure of real ROP exploits. Stage 1 uses ROP to bypass DEP, allowing Stage 2 to execute.

Table 4.1: Estimated number of gadgets in various environments to build a chain to `VirtualProtect`. Null bytes means that the environment allows the injection of null bytes. Static arguments means, that the arguments are known to the attacker beforehand. Arguments passed determines whether parameters are passed in registers or on the stack.

Null bytes	Static arguments	Arguments passed	# of gadgets
yes	yes	stack	0
yes	no	stack	5 - 15
no	yes	stack	0 - 10
no	no	stack	5 - 100
yes	yes	registers	5
yes	no	registers	10 - 25
no	yes	registers	0 - 20
no	no	registers	15 - 100

tacker, and `lpAddress` and `lpflOldProtect` will not change between runs, hence can be determined statically.

Table 4.1 shows rough estimates for how many gadgets are required in different environments with the goal of invoking an API such as `VirtualProtect`. The numbers are by no means set in stone and should mostly show which scenarios are favourable for an attacker and under which circumstances ROP is more difficult. Take, for example, an environment where null bytes are allowed, the arguments are static and passed in registers. Figure 4.2 shows what a simple ROP chain for this scenario on Windows would look like. It consists of very simple gadgets which load parameters directly from memory into the corresponding registers, invokes `VirtualProtect`, and then returns into shellcode. Of course this is an ideal case, which assumes all necessary gadgets are present in the loaded libraries, and the environment places no restrictions on ROP development. While this is true for current environments, there is a lot of research on mitigations that implement certain restrictions.

In the next chapter we discuss what such restrictions might look like and introduce a realistic example for a restriction.

...	
@Gadget1	pop rcx ; ret
data	Parameter lpAddress
@Gadget2	pop rdx ; ret
1000h	Parameter dwSize
@Gadget3	pop r8 ; ret
40h	Parameter flNewProtect
@Gadget4	pop r9 ; ret
data	Parameter lpflOldProtect
@VirtualProtect	
@Shellcode	
...	

Figure 4.2: A simple ROP chain to `VirtualProtect`. Parameter `flNewProtect` has to be set to `40h` which grants read/write/execute permissions.

4.2 Environment Restrictions

We have discussed ROP both in theory and in practice in Chapter 2.4.1 and Chapter 4.1. Usually, short gadgets are combined in such a way, that the `ret` instruction, the last instruction of each gadget, ensures that all gadgets execute consecutively. This leads to ROP often exhibiting traits extremely uncharacteristic in correct program behaviour, such as

- `ret` instructions not returning to an instruction preceded by a `call` instruction
- Large numbers of consecutive short basic blocks
- Considerably more `ret` than `call` instructions

Currently, the environment, i.e., hardware and software, does not implement rules to enforce such basic properties, which allows an attacker to freely combine gadgets in arbitrary ways [1].

In the remainder of this chapter we present a restriction called ROPocop. ROPocop consists of two mitigation techniques, AntiCRA and DEP+. AntiCRA monitors program execution and detects unusually high numbers of short basic blocks. Since it does not rely on `ret` instructions it can detect all kinds of code-reuse attacks, such as JOP, too. DEP+ enforces the concept of non-executable data in software. We implemented ROPocop using PIN [98], a dynamic binary instrumentation tool by Intel. ROPocop is available for download on the companion website[2].

4.2.1 AntiCRA

When designing AntiCRA, we manually analysed code-reuse exploits by looking at the gadgets they use and their properties. We found that the exploits share properties which are unusual and typically not present in a normal program's execution. Based on these observations, we implemented a heuristic which monitors the following two properties:

Indirect Branches

Code-reuse attacks consist of gadgets which all end in an indirect branch. We analysed benchmarks as well as real-world applications like Adobe Reader, VLC, Microsoft Office, Open Office (the complete list can be found in Table 4.2) and found that executing a very high number of consecutive indirect branches is unusual. The highest number of subsequent indirect branches we found during our experiments was 47 (in Microsoft Word), but only 8 of the 35 programs execute 15 or more subsequent indirect branches.

[1] Intel CET [85], which will be part of future CPU generations, will enforce CFI, using, among other techniques, a hardware shadow stack.

[2] `https://sites.google.com/site/ropocopresearch/`

Average Length of Basic Blocks

To reduce side-effects on other registers, the stack, or flags, exploit developers try to use gadgets that are as short as possible. Therefore, at least for contemporary approaches, gadgets can be considered basic blocks with very few instructions. As with indirect branches, we analysed program behaviour of legitimate programs and found that the average number of instructions over a sliding window of 10 basic blocks did not drop below 2.33. Our experiments showed, that making the window smaller resulted in an increase of false positives, while increasing the window resulted in missed attacks. We also found an interesting correlation between this and the previous property: the more consecutive indirect branches, the longer the corresponding basic blocks. We make use of this knowledge in the next paragraph, when we try to find default parameters which work for a wide set of applications.

Since programs exhibit varying characteristics regarding these two properties, ROPocop first runs in learning mode. This requires nothing from the user but simply using the program she wants to protect as usual, while in the background, ROPocop observes the program flow and determines appropriate thresholds for these two properties. This, of course, leads only to limited coverage, however for our approach high coverage is not required. Exploiting a buffer overflow requires some sort of input, generally provided by the attacker as a file that has to be opened by the victim and is then processed by the vulnerable program. Thus, a user working with the program might not cover all possible paths, but it covers the important paths which lead to exploitation.

While we do recommend setting individual thresholds for different programs, at the same time we evaluate whether it is possible to provide default values which cover as many programs as possible. After analysing our test set of benign applications, by running the learning mode and using the programs in our sample set (e.g., opening various media files using VLC, opening various PDF files with Adobe Reader, working with Microsoft Word, etc.) we set the following thresholds: 35 subsequent indirect branches and an average basic block length of 2.25 or lower; as described earlier, we found a correlation that larger numbers of subsequent basic blocks also means longer basic blocks. Therefore we added another threshold; 36 till 50 subsequent indirect branches and an average basic block length of 4 or lower. AntiCRA signals an exploitation attempt if one of the two bounds is violated or if, at any point, more than 50 subsequent indirect branches are executed. While our sample set of benign applications may not be large enough to make a claim, that these suggested thresholds hold for all programs, they do hold for all programs in our set, which includes some of the most exploited applications. Therefore, they serve as an excellent starting point for fine-tuning, should it be required. Since we included many programs that are often found and exploited in business environments (e.g., Word, Excel, Adobe Reader), ROPocop can be deployed immediately without the need to fine-tune thresholds. Programs that make

heavy use of dynamically generated code such as browsers using Javascript, are challenging to protect using AntiCRA. This is because every website using Javascript exhibits different behaviour, making it difficult to find suitable thresholds. This is a problem shared by all heuristic approaches and we leave this challenge for future work.

To increase performance and make the algorithm less prone to false positives, calculating averages starts only after we have collected 15 basic-block lengths, i.e., the first computed average is available only after 15 subsequent indirect branches. This prevents false alarms based on short sequences of short basic blocks, whose sample size is otherwise not significant enough. While this allows for attacks requiring less than 15 gadgets to go undetected by AntiCRA, it decreases false positives because in our experiments we discovered sequences of three to seven consecutive indirect branches with small basic block lengths which would trigger a false alarm. However, these parameters may be changed by the user. Algorithm 1 summarizes AntiCRA in a formal way. Figure 4.3 (in Chapter 4.2.4) shows how the two thresholds form a (shaded) area in a two-dimensional plain. If an execution falls into the shaded area then AntiCRA will signal it as malicious. The figure also summarizes the results of our empirical evaluation, and will be explained in more detail in Chapter 4.2.4.

4.2.2 Impact on Exploit Development

For a code-reuse attack to circumvent AntiCRA, it must not use more than 34 / 49 consecutive indirect branches. If this is possible at all depends on the availability of gadgets, which varies between programs based on what libraries are loaded and whether or not ASLR is employed. Furthermore, the average number of instructions in the gadgets used must never fall below 2.25 / 3.5. Combined, these restrictions make it difficult for an attacker to create a pure ROP or JOP payload. Attackers could attempt to raise the average number of instructions per gadget by inserting longer gadgets. But longer gadgets usually have unwanted side-effects, like manipulating other registers that hold important data, or the stack, or modifying flags. Furthermore, since the total number of gadgets is limited to 34 / 49, inserting long gadgets whose side effects are irrelevant just for the sake of increasing the average wastes precious slots for useful gadgets. To bypass AntiCRA, an attacker would have to try and insert direct branches, but, due to limited availability and side-effects, e.g., potentially losing control over the program counter, this is difficult. Furthermore, we know of no gadget compiler that supports direct branches at this point and have not found exploits that use gadgets that incorporate direct branches. Depending on the program it might still be possible, but, as previously mentioned, AntiCRA's goal is to break current exploits and make the development of new code-reuse exploits significantly more difficult, which AntiCRA certainly achieves.

```
Input: bbl: a basic block, delivered automatically by PIN, thrStartAvgCalc: the
       number of consecutive indirect branches after which the calculation of average
       basic block length starts (default: 15), thrAlarm: the threshold for the average
       basic block length (default: 2.33)
Output: state: a flag that indicates that a ROP attack is probably in progress
state ← noAlarm
cntIndBranch ← 0
avg ← 0
if bbl was reached through indirect branch then
    cntIndBranch ← cntIndBranch + 1;
    log size of bbl;
    if cntIndBranch >thrStartAvgCalc then
        avg ← average length of the last 10 bbl;
        if avg >thrAlarm then
            state ← alarm;
        end
    end
else
    cntIndBranch ← 0;
end
```

Algorithm 1: Algorithm for AntiCRA. We use PIN to continuously monitor basic blocks at runtime and use them as input for our algorithm. Once enough basic block lengths have been collected, i.e., $thrStartAvgCalc$ is exceeded, calculation of average basic block length starts. If this average falls below $thrAlarm$, an alarm is raised.

However, due to its heuristic nature, false positives as well as false negatives are possible. As we show in this work, however, in practice the heuristic seems effective enough to go without any false decisions, at least in our benchmark set. Furthermore, under circumstances very favourable to an attacker it might be possible to create a two-staged exploit that disables DEP using fewer than 15 gadgets and then runs a regular payload. This would not be detected by AntiCRA.

4.2.3 DEP+

DEP+ is based on the same concept as DEP, i.e., the premise that data should not be executable. DEP+ thus monitors the loading and unloading of images and creates a virtual memory map based on this information. All virtual memory space where no image is mapped is considered to hold potentially malicious data, since stacks or heaps can be allocated in these areas. To enforce that the instruction pointer never points outside an image, DEP+ checks the register's value after each indirect branch, i.e., after each return, indirect call, and indirect jump. Opposed to DEP, DEP+ cannot be bypassed through API calls such as `VirtualProtect`.

Implementation Details

PIN's `IMG_AddInstrumentFunction` as well as `IMG_AddUnloadFunction` are used to monitor the loading and unloading of images. When an image is loaded, DEP+ stores its start and end address; if the same image is unloaded at runtime, this information is removed. This approach results in a virtual-memory map that distinguishes only between images and non-images, i.e., code regions and data regions. DEP+ treats these data regions as space for potentially malicious data, hence does not allow `rip` to point into it. To do so, DEP+ checks after any indirect branch is taken, but before it is executed, if the instruction pointer points inside any of the data regions. Algorithm 2 summarizes DEP+ in a formal way.

The reason DEP+ checks if `rip` points inside data regions instead of checking if `rip` points inside a loaded image is due to performance: many programs load 30 or more libraries, which means that there can be an equally high number of code regions that need to be checked. As we found, checking each of those regions after each indirect branch can incur a significant performance penalty. To increase performance on Windows, we thus make use of the fact that Windows' memory management is relatively deterministic. Images, in general, tend to be loaded at very high addresses, around `0x60000000` and higher, while stacks and heaps reside at low addresses and new ones are allocated towards increasingly higher addresses. Depending on the memory usage of a process, it is generally valid to assume that stacks and heaps, where an attacker would inject his payload, reside below most images.

Input: p: a program, $mmap$: a map of virtual memory that contains start and end addresses of loaded images and allows calculating start and end addresses of data regions

Output: $state$: a flag that indicates that code is executed from outside an image

$state \leftarrow noAlarm$

if *instruction pointer points inside a data region in mmap* **then**

| $state \leftarrow alarm$

end

Algorithm 2: Algorithm for DEP+ (without performance optimizations). We use PIN to instrument the target program so the algorithm is called after an indirect branch is taken, but before the instruction is executed. It checks whether the instruction pointer points into a data region, i.e., outside all loaded images.

DEP+ makes use of this knowledge by not checking if `rip` points inside any of the loaded images but instead checking if `rip` points inside data regions, where heaps and stacks are located, which results in a much lower number of necessary checks. To this end, DEP+ monitors a program's heap and stack sizes to dynamically increase or decrease the number of data regions that need to be taken into account.

This is, of course, a heuristic, which trades security for performance, but as our evaluation in Chapter 4.2.4 shows, the heuristic helps DEP+ to bring the checks down to a minimum while still recognizing all tested attacks.

Comparison to DEP

The original shortcomings of DEP are that it may not be enabled at all, or that it can be bypassed by both pure code-reuse attacks and by code-reuse attacks that invoke an API such as `VirtualProtect` to disable DEP. DEP+ improves over DEP in that it prevents the execution of injected code by enforcing non-executable data regions *even* for processes that run with regular DEP disabled. In particular, DEP+ cannot be bypassed by calls to operating system APIs like `VirtualProtect` and its siblings, as such calls have no effect on DEP+. Hooking said functions is not an option either, because the operating systems regularly uses legitimate calls to these functions, e.g., when loading executables. Furthermore, as Chapter 4.2.4 shows, the overhead introduced by DEP+ is negligible.

Limitations

Processes which rely on the ability to execute code from outside images, e.g., processes which generate code at runtime or incorporate self-modifying code, are not compatible with DEP+. Such a process is not compatible with DEP either, unless it uses the `VirtualProtect`

API etc. to disable DEP for memory regions with generated code. Since it is difficult to detect whether a call to the API usually abused to bypass DEP by an attacker is legitimate, i.e., originating from the program itself, we decided against supporting such calls. This results in a strong increase in security, at the drawback of reduced compatibility.

Like DEP, DEP+ cannot detect and thus not prevent the exploitation of the vulnerability itself, e.g., the overwriting of data on the stack due to a buffer overflow. Therefore, non-control data attacks [30] or information leakages are still possible. Furthermore, DEP+ does not prevent *pure* code-reuse attacks, motivating the need for AntiCRA (Chapter 4.2.1).

4.2.4 Evaluation

Our implementation is highly modular, so that one may deploy AntiCRA or DEP+ independently as well as in combination. Running both of them, however, strongly increases security, in a similar fashion as running with DEP and ASLR.

In this chapter we evaluate AntiCRA and DEP+ by addressing the following research questions:

RQ1: How effectively does AntiCRA detect pure code-reuse payloads?

RQ2: How effectively does AntiCRA detect two-staged ROP payloads?

RQ3: How effectively does DEP+ detect code-injection attacks?

RQ4: What is the performance overhead of AntiCRA and DEP+?

Evaluation of AntiCRA (RQ1/RQ2)

For evaluating RQ1 we looked at pure code-reuse attacks, however, at this point such payloads are only rarely found in the wild and are mostly used in academia as proof of concept. The only real-world pure code-reuse exploit we found is a ROP exploit for Adobe Reader. Since neither the exploit's source code, nor an infected file are publicly available, our conclusion is based on an analysis by Li and Szor [95]. Analysing the exploit's source code reveals that the address `0x6acc1049` is repeated 9,344 times; the instruction at that address is a simple `ret`. This equals to over 9,000 indirect branches in a row, which would, of course, be detected by AntiCRA.

The likely reason for why pure ROP and JOP payloads still seem to be rare in practice is that two-staged payloads (which aim to disable DEP through ROP/JOP) are simpler to construct and are sufficient in many cases.

We analysed 11 real-world exploits in total. To operate on an unbiased test set, we analysed the 10 most recent exploits from `http://www.toexploit.com/`[3] which claim to bypass ASLR and also added the previously mentioned pure ROP exploit. Figure 4.3 and Table 4.2 show the results of our analysis, i.e., the number of consecutive indirect branches and the average basic block length for each exploit and also for legitimate programs. As the numbers indicate, legitimate programs rarely have more than 15 consecutive indirect branches and their average basic block length is higher than that of exploits. This confirms that our generalized thresholds, which work for a wide variety of programs, are well-suited to detect attacks.

AntiCRA detects 10 out of the 11 exploits in our sample set. In five cases this is due to the number of indirect branches in a row. Three exploits are detected because they use very short gadgets, which mostly only execute one instruction and then transfer program execution to the next gadget. Two exploits trigger both mechanisms, since they use more than 35 indirect branches in a row and also very short gadgets.

One exploit cannot be detected by AntiCRA. This is because it requires only 13 gadgets to prepare the stack for calling `VirtualProtect`. This is not enough to trigger the indirect-branch check. The average length of the basic blocks is 2.2, which would trigger an alarm. However, as explained in Chapter 4.2.1, we only trigger inspections after a total of 15 indirect branches in a row.

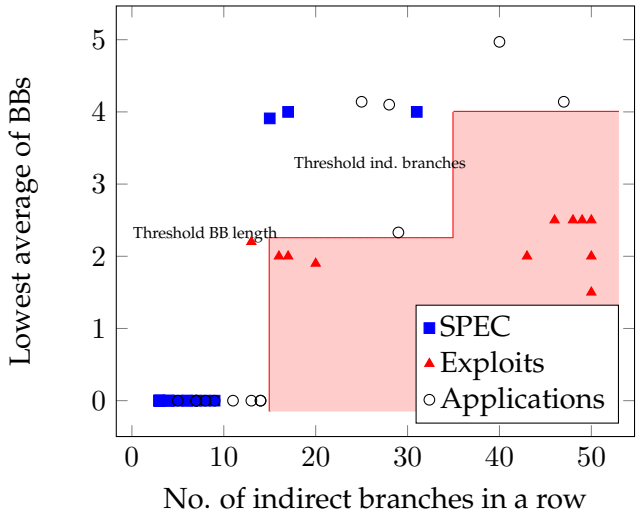


Figure 4.3: Analysis of the number of indirect branches in a row and the lowest average basic block length of our test set

[3]Unfortunately, the website is not available anymore, but the exploits can be requested from the author of this thesis or found online using any search engine and the full name of the exploit, as shown in Table 4.2

ID	Name	Indirect Branches, threshold: 35	Average BB-Length, threshold: 2.25
E01	ASX to MP3 Converter v3.1.2.1 SEH Exploit (Multiple OS, DEP and ASLR Bypass)	>**50**	>2.25
E02	BlazeDVD 5.1 Stack Buffer Overflow With ASLR/DEP Bypass	20	**1.9**
E03	BlazeDVD 6.1 PLF Exploit DEP/ASLR Bypass	16	**2**
E04	DVD X Player 5.5.0 Pro / Standard version Universal Exploit, DEP+ASLR Bypass	17	**2**
E05	DVD X Player 5.5 Pro (SEH DEP + ASLR Bypass) Exploit	13	(2.2)*
E06	ProSSHD 1.2 remote post-auth exploit (w/ASLR and DEP bypass)	**43**	**2**
E07	RM Downloader 3.1.3 Local SEH Exploit (Win7 ASLR and DEP Bypass)	**49**	>2.25
E08	The KMPlayer 3.0.0.1440 .mp3 Buffer Overflow Exploit (Win7 + ASLR bypass mod)	**46**	>2.25
E09	UFO: Alien Invasion v2.2.1 BoF Exploit (Win7 ASLR and DEP Bypass)	>**50**	>2.25
E10	Winamp v5.572 Local BoF Exploit (Win7 ASLR and DEP Bypass)	>**50**	**2**
E11	Adobe Reader 11.0.01 “Number of the Beast” (ASLR, DEP, Sandbox bypass, pure ROP)**	>**50**	-
E12	QQ PLAYER PICT PnSize Buffer Overflow WIN7 DEP ASLR BYPASS	11	(2)*

A01	Daemon Tools v. 4.47.1.0333	40	4.97
A02	Microsoft Word 2010 v. 14.0.07140.5002	47	4.14
A03	Microsoft Excel 2010 v. 14.0.07140.5002	28	4.1
A04	Microsoft Powerpoint v. 14.0.07140.5002	25	4.14
A05	Adobe Acrobat Pro v. 9.0	29	2.33
A06	Windows Media Player v. 12.0.7601.18150	11	na*
A07	cmd.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	5	na*
A08	calc.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	6	na*
A09	mspaint.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	13	na*
A10	taskmgr.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	9	na*
A11	VLC v. 2.0.8	9	na*
A12	Irfanview v. 4.3.3	12	na*
A13	Notepad++ v. 6.1.4	11	na*
A14	Filezilla v. 3.5.3	7	na*
A15	Open Office Writer v. 4.0.1	7	na*
A16	Open Office Impress v. 4.0.1	7	na*
A17	Open Office Calc v. 4.0.1	8	na*
B01	SPEC CPU2006 - 400	4	na*
B02	SPEC CPU2006 - 401	3	na*
B03	SPEC CPU2006 - 403	6	na*
B04	SPEC CPU2006 - 429	3	na*
B05	SPEC CPU2006 - 433	3	na*
B06	SPEC CPU2006 - 444	4	na*
B07	SPEC CPU2006 - 445	5	na*
B08	SPEC CPU2006 - 447	7	na*
B09	SPEC CPU2006 - 450	8	na*
B10	SPEC CPU2006 - 453	6	na*
B11	SPEC CPU2006 - 456	3	na*

B12	SPEC CPU2006 - 458	3	na*
B13	SPEC CPU2006 - 464	31	4
B14	SPEC CPU2006 - 470	4	na*
B15	SPEC CPU2006 - 471	15	3.91
B16	SPEC CPU2006 - 473	3	na*
B17	SPEC CPU2006 - 482	9	na*
B18	SPEC CPU2006 - 483	17	4

Table 4.2: Analysis of Exploits and Programs. We assigned a unique ID to every exploit and program we investigated. The last two columns show our metrics for AntiCRA, i.e., the highest number of indirect branches taken and the average basic block length. Bold numbers show that AntiCRA was triggered, due to an exceeded threshold. (*not computed by AntiCRA due to low number of indirect branches (<15) **data is based on the analysis by Li and Szor [95])

Evaluation of DEP+ (RQ3)

To test DEP+, we wrote a small vulnerable application, which uses an unbounded `strcpy` and was compiled with the `NX_COMPAT`, and a simple exploit. Since all code injection attacks store the injected code inside a buffer which, by definition, cannot be in an image, the program that contains the vulnerability is of little consequence. The only differences between our vulnerable application and a real application are mitigation techniques which might be in place, but which are irrelevant to us, since we assume an attacker is able to bypass them, and how program flow is transferred to the injected code, which is irrelevant for our evaluation as well. Ultimately, all code injection attacks end up calling their injected code, and this is where DEP+ detects them. Therefore, evaluating DEP+ with this self-written program poses no real threat to the validity of this experiment. As expected, DEP+ correctly detects that the target address of the `ret` instruction at the end of our vulnerable function is not in an image, before the instruction is actually executed. Therefore, it can terminate the program and mitigate an attack, which would have led to arbitrary code execution. As for the real world exploits, DEP+ detects each one except for the pure ROP exploit for Adobe Reader, as all the others eventually do execute code from memory outside of images.

Performance (RQ4)

We evaluated the performance of ROPocop using the C and C++ benchmarks in the SPEC CPU2006 benchmark suite. Note that those are really worst-case benchmarks that exercise

the dynamic analysis heavily. Any interactive or network-based application would show a significantly lower overhead. We measured five different runtimes for each benchmark:

- The native runtime, i.e., without PIN.
- The runtime with PIN attached, but without instrumentation, to get the basic overhead PIN introduces.
- The runtime with AntiCRA.
- The runtime with DEP+.
- The runtime with AntiCRA and DEP+.

Benchmarks were run on Windows 7 SP1 with an Intel Core 2 Duo T9400 clocked at 2.53 GHz and 4 GB RAM using the reference workload.

Figure 4.4 summarizes the results of our performance benchmarks. Running a program under PIN but without any instrumentation introduces an average overhead[4] of 1.36x, i.e., programs take, on average, 36% more time to finish, ranging from 1.002x (470.lbm) to 2.24x (464.h264ref). Programs protected by AntiCRA run, on average, with a total overhead 2.2x. With DEP+ enabled as well, ROPocop introduces an average overhead of 2.39x, which is comparable to similar tools such as ROPdefender [50], which gives weaker guarantees.

While overheads in the order of two-fold might sound unacceptable, those overheads should really only be expected in worst-case situations. Benchmarks like SPEC CPU2006 are designed to stress CPU and memory, but what ultimately counts is the performance on real-world applications. Their performance can, however, often hardly be measured systematically, which is why we only report qualitative results on some of the applications in our sample set. As a general observation we can say that in all cases the GUI had some slight input lag < 1 second when opening a menu for the first time, however, afterwards they opened in an instant. File transfers with Filezilla were no slower than without our tool. VLC plays h.264 encoded HD videos without any jitter. Adobe Reader renders pages without any noticeable lag. Typing in Microsoft Word has no input lag. We want to emphasize that ROPocop is not intended to be used with all applications at all times. Instead, our recommended usage is to enable it only for either very critical systems, or for an application which has a vulnerability that is being actively exploited and no vendor patch has been released yet. Under such circumstances the overhead is, in our opinion, acceptable.

[4] Average overheads were computed using the geometric mean, which is considered best practice for reporting normalized values such as percentages of overhead [61]

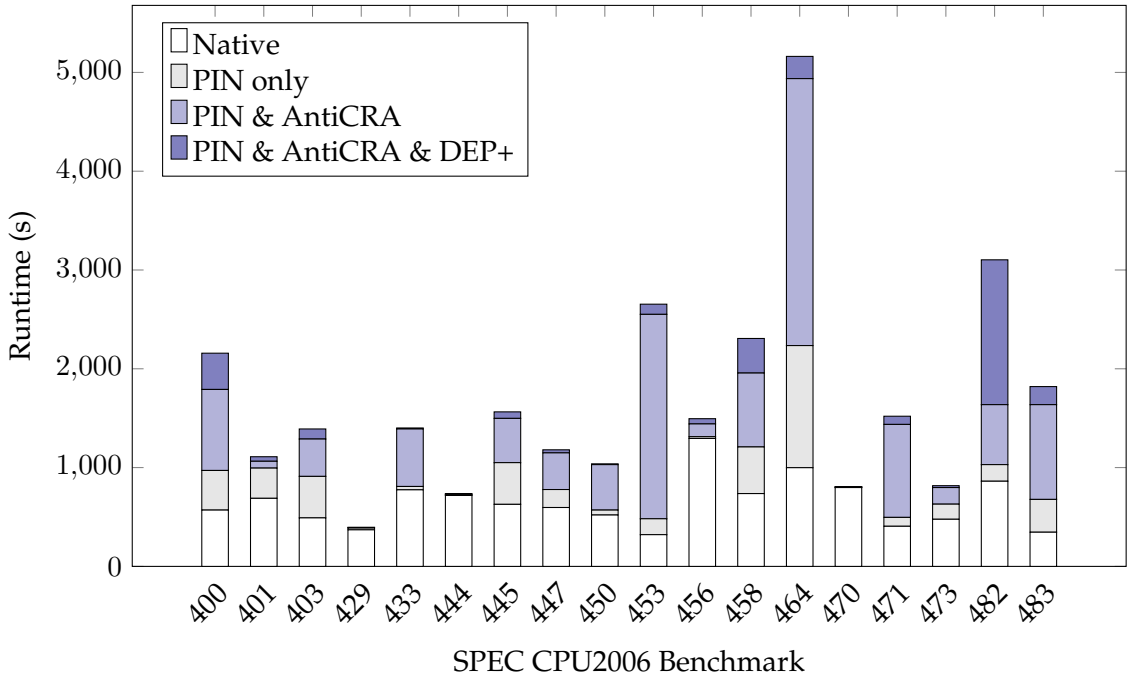


Figure 4.4: Performance of ROPocop

4.2.5 Summary

This chapter presented ROPocop which consists of DEP+ and AntiCRA. DEP+ enforces a non-executable stack and heap in software. AntiCRA is a set of heuristics designed to detect program behaviour normally exhibited by ROP exploits. It monitors program execution, counting the number of consecutive indirect branches and calculating the average number of instructions in basic blocks. Configurable thresholds allow the user to fine-tune the thresholds to individual programs, further increasing security. While having a relatively high overhead, ROPocop is very effective at detecting basic ROP exploits.

Restrictions such as AntiCRA drastically complicate ROP exploit development. An analyst has to carefully craft a gadget chain that has to contain longer and therefore more complex gadgets. This makes manually keeping track of aliasing, side effects, and preconditions difficult. Automating this process would provide a great benefit and allow the analyst to invest more time in other areas of exploit development, such as bypassing mitigations, e.g., ASLR or stack canaries. The next chapter discusses possibilities and limits of gadget chain automation.

4.3 Automated Gadget Chaining

With restrictions imposed by the environment, such as ROPocop, manual exploit development gets increasingly difficult. But even without restrictions, as it is currently the case,

chaining gadgets together manually is a menial task. Many tools which support this process exist, however, their usefulness is usually limited as their main purpose is gadget discovery. While some employ rudimentary auto-chaining capabilities, they often fail, even when the necessary, simple gadgets are available. In conclusion, the current state of gadget chain automation allows for major advancements.

One big issue regarding automation is that the plurality of mitigation techniques, such as DEP, ASLR, and stack canaries require different techniques to be bypassed. DEP is bypassed using ROP, which is a universal bypass. However, for bypassing ASLR or stack canaries, a large number different techniques exist, which apply to different scenarios. This makes full automation nearly impossible.

Another large challenge automation has to overcome is state explosion: assuming the exploit developer wants to call a Linux function that takes six arguments, and there are 15 different gadgets for loading each register, this results in $15^6 \times 6! = 8,201,250,000$ possible combinations. This can actually be seen as an example of a good case, as 15 different gadgets for each register is on the lower side. Clearly, manually evaluating whether the result of an arbitrary combination is the desired one is not feasible. But even with growing computational power, analysing billions of complex gadget chains is not realistic.

In the remainder of this chapter we present PSHAPE, a framework that supports exploit developers in the later steps of exploit development and automates parts of ROP chain generation.

4.3.1 PSHAPE

PSHAPE assists an analyst during exploit development by offering two distinct features which set it apart from existing gadget finders or auto-roppers that are publicly available, namely it (i) provides useful summaries based on gadget semantics, making it straightforward for an analyst (or tool) to assess and select gadgets, and (ii) chains gadgets together so that they load registers used to pass parameters to functions with analyst-controlled data. This allows the invocation of arbitrary functions. PSHAPE also ensures that any preconditions of a gadget (such as that a register has to point to readable memory) are satisfied.

We first define what gadget summaries are and how they are computed in Chapter 4.3.2. Then, we describe our approach to generate gadget chains in Chapter 4.3.3.

4.3.2 Gadget Summaries

Overview ROP mitigations that (i) monitor program executions and detect short code sequences [33, 50, 64, 120] or (ii) require all `return` operations to return to an instruction following a call instruction [1, 120, 174, 175] force developers into using longer and longer gad-

gets, or even entire functions [138]. The increasing length of gadgets makes manual analysis and reasoning increasingly difficult. We thus propose *gadget summaries*, which reflect a gadget's semantics in a compact specification that allows analysts to understand a gadget's behaviour at a glance. Figure 4.5 shows an example of a gadget summary, with the gadget on the left, and its summary on the right[5]. This gadget has two preconditions, because `r9` and `rsp` are dereferenced. The actual effects on the program state are that `rsp` is increased by 8, `rax` receives the value of 1 + `[r9 + 4]`, and `rcx` is assigned the value of `r9`.

Method The process of producing summaries is as follows. First, gadgets are identified by finding `return` opcodes and backward disassembly. These gadgets are then converted into an *intermediate representation* (IR) to simplify analysis. Our current prototype uses VEX IR, see Chapter 4.3.4.

Based on this IR, PSHAPE propagates all assignments, such as to temporary or real registers, or memory locations forwards, resulting in a single statement for each real register and memory location. This single statement (referred to as *postcondition*) contains all operations on this register or location, i.e., an abstraction of the new value after a gadget has executed. This also allows us to readily extract *preconditions*, such as register or memory dereferences. Post- and preconditions combined result in a *gadget summary*, a compact representation of the state of memory and registers after a gadget has executed along with a list of dereferenced registers and offsets. Our syntax for pre- and postconditions is similar to assembler syntax, and should be intuitive for binary analysts. The current prototype excludes instructions such as jumps, loops, or bit manipulation in the summaries to reduce the explosion in state and complexity. We leave more involved search strategies for future work.

As memory is often accessed sequentially using offsets from the same register, one can compress summaries by merging such accesses into a range. For example, preconditions `[rax]`, `[rax + 8]`, `[rax + 0x10]` and `[rax + 0x20]` can be compressed to:

```
[rax] <-> [rax + 0x20]
```

This denotes that all memory between `[rax]` and `[rax + 0x20]` has to be read/writeable. This heuristic sacrifices precision, as not every single byte must be accessed, but helps in making summaries concise.

Gadget summaries aid the analyst in the process of understanding how a gadget affects the state of registers and memory and are increasingly helpful, the more instructions and aliasing a gadget contains. They allow for a more efficient gadget search, as expressing postconditions when searching for a gadget is much more intuitive and flexible than specifying a certain instruction. Think, for example, of loading 8 bytes from memory, pointed

[5]The fact that a reviewer erroneously stated that the summary was wrong actually shows how error-prone doing this manually is. This reaffirms that our summaries are of great help.

```
mov rax, rsp
mov [rax+20h], r9
mov [rax+18h], r8
mov [rax+10h], rdx
mov [rax+8], rcx
mov rcx, r9
mov rax, [rcx]
inc rax
mov [rcx+8], rax
mov rax, [rcx+4]
inc rax
mov [rcx+0Ch], rax
ret
```

(a) A candidate gadget.

```
PRE: [r9] <-> [r9 + 0xC]
PRE: [rsp] <-> [rsp + 0x20]
POST: rsp = rsp + 8
POST: rax = [r9 + 4] + 1
POST: rcx = r9
```

(b) Gadget summary for 4.5a.

Figure 4.5: Despite this being a relatively short gadget in `mshtml.dll` which contains only 13 instructions (a), analysing it manually is still a cumbersome and error-prone task. PSHAPE automates this process by creating a simple summary (b). Note that by default PSHAPE does not display memory write postconditions as they are seldom of interest, and make the summary harder to read.

to by `rsp`, into `r8`. An obvious way of searching for such a gadget is looking for a `pop r8` instruction. However, `mov r8, [rsp]`, or `pop rax # mov r8, rax`, or `pop rax # xchg rax, r8`, and many others work as well. Having to search for all semantically equivalent instruction sequences is tiresome and error-prone. Using summaries, the analyst can simply search for the postcondition `r8 = [rsp]`. This ability is extraordinarily useful when the analyst needs gadgets that fulfill a specific purpose, for example, a stack pivot gadget, or a gadget that manipulates memory, such as a write-what-where gadget. Lastly, gadget summaries are useful for selecting gadgets for automated gadget chain generation, which we describe in the next chapter.

4.3.3 Gadget Chaining

Our approach aims at finding a valid and short gadget chain which loads analyst-controlled data, i.e., relative to `rsp`, into registers. This allows invoking an arbitrary function with analyst-specified parameters. It consists of three steps, as shown in Figure 4.6. In the first

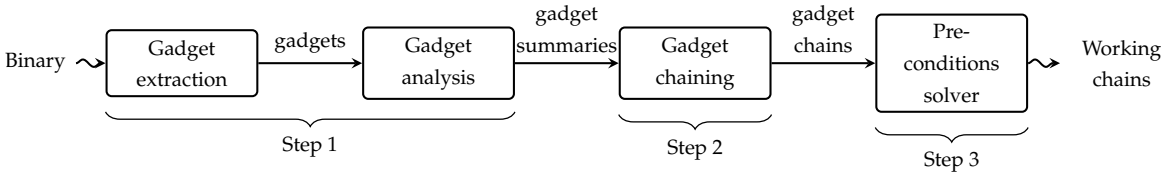


Figure 4.6: Overview of our approach to generate gadget chains. In Step 1, we extract gadgets up to a certain size and create summaries, selecting those that can be used to load m parameters into registers. In Step 2 we select a set of n gadgets for the individual parameters to constrain the search space. In Step 3 we search if gadgets can form a chain by permuting and analysing the available gadgets.

step, PSHAPE extracts gadgets from the target binary and computes summaries. Then, based on the summaries, it filters the list of gadgets to keep only the ones related to initializing registers that are used for passing function parameters. The second step combines these gadgets into chains. For a chain, pre- and postconditions are computed, and if the chain has the desired postconditions, the third step commences. In this step, PSHAPE analyses the validity of each chain is and adds gadgets to satisfy any preconditions.

Step 1: Gadget Extraction and Summary Computation. First, PSHAPE extracts gadgets from a given binary. This step yields a list of gadgets for which we then compute gadget summaries. The results are stored, making them available for the analyst.

Next, it filters gadgets to keep only the ones related to initializing registers used for passing parameters to functions. On 64-bit Windows those are `rcx`, `rdx`, `r8`, and `r9`, in that order. On 64-bit Linux the registers used for parameter passing to functions are `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, in that order. Additional parameters are passed on the stack in both cases. Our summaries simplify filtering, as gadgets that do not set the registers stated above to a value that can be controlled by the analyst are discarded automatically.

We divide these gadgets into two categories, *load* and *mod*. Gadgets in the load category overwrite a given register, e.g., a `pop` instruction, while gadgets in the mod category modify it, e.g., an `add` instruction. Gadgets in the load category are favoured, and within this category, gadgets that use `rsp`-relative memory dereferences are preferred, as `rsp` needs to be under the control of the analyst anyway when using ROP. For example, a `pop rcx` gadget is preferred over a `mov rcx, [rax]` gadget. This is because the former gadget uses `rsp`, which is analyst-controlled, as opposed to the latter gadget, which uses `rax`, which may or may not be analyst-controlled. Based on this ranking and the number and severity of pre-, and postconditions, the n most suitable gadgets for loading each parameter register with arbitrary data are selected and passed to Step 2. We present details about gadget selection in Chapter 4.4.

Step 2: Combining Gadgets into Chains. In the second step, the gadgets from Step 1 are combined and all possible permutations are computed. Remember that in Step 1, n gadgets are selected for every parameter register. Assuming a function that takes m parameter registers is invoked, this results in $n^m \times m!$ possible chains. For each chain, pre- and postconditions of the whole chain are computed, and if a chain's postconditions are not the expected result, i.e., the registers used to pass parameters do not contain user-controlled data, it is discarded. Instead of exhausting the search space, we stop the exploration after the first viable combination is found. Strategies other than this generate and test approach are certainly possible, however, our vast reduction of the search space by grading gadgets, allows us to use this approach and find a solution in a timely fashion.

Step 3: Solving Pre-Conditions. The third step solves preconditions. Indeed, it may happen that a chain generated in Step 2 contains preconditions such as register dereferences, meaning that the analyst needs to have the possibility to initialize the dereferenced registers, so they contain the address of a valid memory area. In Step 3, PSHAPE attempts to build a gadget chain that allows loading user-controlled data into an arbitrary register. Once such a gadget is found it is prepended to the incoming chain, forming a new chain. The new chain is then checked for pre- and postconditions again to make sure it does indeed initialize dereferenced registers and does not interfere with the original chain. Note that the number of iterations is limited (four in our prototype), so the chain does not grow forever. This could happen if a gadget which is added to solve a precondition, itself adds a new precondition.

Our gadget chaining fully automates the process of stitching gadgets together to initialize registers used for passing parameters to functions with data the analyst controls. It also adds gadgets to the chain to ensure any dereferenced registers are also initialized with data the analyst controls. This approach simplifies exploit development, especially if functions taking many parameters are called or if the available gadgets consist of many instructions.

4.3.4 Implementation

PSHAPE uses a standard technique to discover gadgets: first, using pyelftools [11] and pefile [27], it finds executable sections in an input binary. Afterwards it scans these sections forwards byte by byte until it finds a return opcode, and stores these offsets in a list. Then, using several threads, it disassembles backwards from these offsets using the Capstone framework [112]. To limit the number and complexity of gadgets and to speed up the discovery process, the analyst can specify the minimum and maximum size, i.e., number of instructions, of a gadget. Please note that we keep disassembling backwards for 5 more bytes (a configurable heuristic), even when the maximum gadget length has been reached,

or if, e.g., a `ret` instruction is encountered. Since the disassembly can change depending on preceding bytes, this helps us discover more gadgets.

If the disassembly is successful, i.e., yields only legal instructions, we convert, or *lift*, this gadget to Valgrind's VEX IR [111] using PyVEX [146]. Lifting the original assembly code to VEX has the advantage that it is much simpler to analyse because there are fewer instructions and side effects are made explicit. After this conversion, VEX assignments are propagated forward, resulting in a single statement for each real register and memory location, which contains all operations on this register or location, i.e., an abstraction of the new value after a gadget has executed.

In the next chapter, we evaluate PSHAPE, the implementation of our approach, and discuss three gadget chains that it created fully automatically.

4.3.5 Comparison with Existing Tools

In Table 4.3 we have listed the tools designed to help an analyst to create ROP exploits. All tools extract gadgets from a given binary (syntactic search, column 1). However, only half compute semantic information from gadgets (semantic search, column 2), i.e., take into account *what* a gadget does, instead of *how* it does it. About 70% state that they can automate gadget chain generation (gadget chaining, column 3). Only one is Turing complete, however, this is not a requirement for ROP development since most actual exploits do not use branching or loops. About 85% are open-source and/or have a binary version publicly available. All but one can handle PE binaries, and all but two can handle ELF binaries. About 70% can handle 64-bit binaries.

OptiROP and Q are not publicly available and also were not made available to us upon request. We also excluded nrop, as its scope is different from a traditional gadget finder: given a certain instruction as input, it only finds gadgets that are semantically equivalent. We managed to compile ROPC although it has been unmaintained for more than three years, and GitHub issue reports are not answered. Unfortunately, it could not extract gadgets from any of the binaries we use in the evaluation, which is why we exclude it.

For our evaluation we use a total of nine binaries. We use five Windows binaries: firefox.exe, iexplore.exe, chrome.exe, mshtml.dll, and jfxwebkit.dll and four Linux binaries: chromium, apache2, openssl, and nginx, representing a total of 147 MiB of executable data. Detailed information about the binaries as well as PSHAPE itself are available on the companion website [6].

[6] `https://sites.google.com/site/exploitdevpshape/`.

Tool	Syntactic Search	Semantic search	Gadget Chaining	Turing Complete	Open-Source	Binary available	PE	ELF	64-bit
PSHAPE	✓	✓	✓	×	✓	✓	✓	✓	✓
OptiROP [113]	✓	✓	✓	×	×	×	✓	✓	✓
nrop [167]	✓	✓	×	×	×	×	✓	✓	✓
Q [140]	✓	✓	✓	×	×	×	✓	✓	✓
ROPC [118]	✓	✓	✓	✓	✓	✓	✓	✓	✓
DEPLib [152]	✓	✓	✓	×	✓	✓	✓	×	×
Agafi 1.1 [66]	✓	×	✓	×	✓	✓	✓	×	×
mona.py 2.0 (rev566) [38]	✓	×	✓	×	✓	✓	✓	×	×
ROPgadget 5.4 [136]	✓	×	✓	×	✓	✓	✓	✓	✓
rp++ 0.4 [154]	✓	×	×	×	✓	✓	✓	✓	✓
Ropeme [53]	✓	×	×	×	✓	✓	×	✓	×
ropper 1.8.7 [137]	✓	×	✓	×	✓	✓	✓	✓	✓
MSFrop [109]	✓	×	×	×	✓	✓	✓	✓	×

Table 4.3: Summary of ROP tools. Note that many tools have limitations regarding gadget discovery and chaining, which we discuss in Chapter 4.3.5 and Chapter 4.3.5, respectively.

Gadget Discovery

Here we compare the different gadget discovery routines. For a tool to be considered in these experiments, we require that it can read ELF or PE binaries and is able to find gadgets in 64-bit binaries. DEPlib, Agafi, mona.py, Ropeme, and MSFrop do not fulfil these requirements and were therefore discarded, leaving us with the following tools to compare to: ROPgadget, rp++ and ropper. We configured them to look for gadgets up to a maximum length of 35 instructions. This is a drastic increase compared to gadgets currently used, which mostly tend to contain only two or three instructions, but at the same time still allows all evaluated tools to finish within a reasonable time frame. Table 4.4 summarizes the results.

ROPgadget works out of the box, but its output contains duplicates, i.e., the same gadget at the same address is listed more than once. We informed the developer about this bug. ROPgadget does not have an option to define the maximum number of instructions in a gadget. Only the maximum number of byte per gadget can be set. We ran our experiments using 110 bytes for the maximum length, leading to an average opcode size of about 3 bytes per instruction. Originally, we planned to use a much larger number to make sure we do not miss any gadgets. However, even with a depth of 110 bytes the evaluation of ROPgadget on Chromium took over 6 hours, consuming 160 GB of RAM. Afterwards, we used a script to go through the results and remove any gadgets that contained more than 35 instructions.

Therefore, we miss gadgets that contain 35 or fewer instructions but are longer than 110 bytes.

rp++ originally comes with a fixed maximum gadget length of 20 instructions. We modified the source code, changing this upper limit to 35 and recompiled it, so it is able to correctly discover longer gadgets, too.

ropper While running small-scale experiments using ropper, we noticed that it does not show call and jump instructions in gadgets it finds. This makes it difficult to use for an analyst, because every gadget would have to be checked for omitted instructions before it can be used. Furthermore, ropper did not find some simple and short gadgets. We informed the developer, who acknowledged the bugs. The updated version (1.8.7) does not omit any instructions in its output. It does, however, still show gadgets containing conditional jumps. Such gadgets are difficult to use, especially since no information is given about which paths are taken under which circumstances.

Since all four tools use slightly different filters or sometimes contain bugs, it is difficult to compare their results. For example, ROPgadget and rp++ keep gadgets that contain privileged instructions (e.g., `in`, `out`, or `hlt`), which likely terminate the process. ROPgadget's output contains duplicate gadgets, and ropper keeps gadgets that contain conditional jumps, which the other tools do not. We filter and clean the output of all tools, removing any duplicates and privileged instructions as well as jumps. As Table 4.4 shows, all tools find a similar number of gadgets. Next, we look into the tools' gadget chaining abilities.

Gadget Chaining

Here, we evaluate the tools in regards to their ability to create gadget chains. As before, our minimum requirements for a tool to be considered in the experiments are that it can build ROP chains for 64-bit Windows or 64-bit Linux, correctly initializing the registers used for passing parameters to functions. We use functions that are regularly used in ROP exploits. For Linux, the goal is to create two chains, one that loads registers with analyst-controlled data for invoking a function that takes three arguments (e.g., `mprotect` or `execve`) and one chain that loads registers with analyst-controlled data for invoking a function that takes six arguments (e.g., `mmap`). For Windows, the goal is to create a chain that loads registers with analyst-controlled data for invoking a function that takes four arguments (e.g., `VirtualProtect` or `VirtualAlloc`). From this point on, we refer to these goals by the function's names but keep in mind that any function using the same number of parameters or fewer can be invoked, too.

From the list of available tools, only ROPgadget and ropper satisfy our requirements. While DEPlib, Agafi, and mona.py, have gadget chaining capabilities, they only work on 32-bit Windows platforms. Since the gadget discovery routine of ropper is flawed, we manually

make sure that all gadgets in its final ROP chain are in fact correct and usable. The results of the experiments have been summarized in Table 4.5.

ROPgadget is not able to create chains for Windows platforms, does not offer any targets for a ROP chain and instead always tries to build a chain to create a shell using `execve`. However, this function requires initializing three arguments, allowing us to evaluate at least one goal for Linux. ROPgadget successfully created a chain for chromium, but it did not succeed on any of the remaining binaries.

ropper cannot create chains for 64-bit Windows, but offers two targets for ROP chain creation on 64-bit Linux, `mprotect` and `execve`, which both take three arguments. Again, this allowed us to evaluate at least one of the goals we specified previously. However, for openssl and nginx, ropper was able to initialize only `rdi`, despite discovering several useful and simple gadgets that load the other registers. For apache2, ropper successfully initialized `rdi` and `rdx`. Ropper successfully created a ROP chain for chromium, initializing all three registers used for passing parameters to `mprotect` or `execve`. All gadgets used in the chains are without side-effects and without preconditions. Thus, no additional work to satisfy preconditions is necessary.

PSHAPE successfully created fully functional chains for both `mprotect` and `mmap` for the following Linux binaries: chromium, apache2, and nginx. We present and discuss the chains for apache2 and nginx in Chapter 4.3.6. For openssl it was only possible to create a chain to `mprotect`. This was due to the fact that no gadget was found to initialize `r9`, which we confirmed manually using both PSHAPE and ROPgadget. On Windows binaries, PSHAPE failed to build chains for firefox.exe and iexplore.exe, and we confirmed, again using both PSHAPE and ROPgadget, that, in fact, the necessary gadgets are not present in the respective binaries. For mshtml.dll and jfxwebkit.dll, PSHAPE successfully built a chain. It also created a chain for chrome.exe, however, it required another gadget to be prepended manually. (details below) Hence, we did not count it towards successful chain creations in Table 4.5. We discuss this chain and its shortcomings in Chapter 4.3.6.

In cases where PSHAPE failed to build a chain, we evaluate whether a human analyst would actually be able to succeed. In other words, we assessed if it was in fact not possible to build a chain, due to a lack of useful gadgets, or if our tool's limitations were to blame. In the case of openssl and iexplore.exe, the former is the case. While there are gadgets that initialize the registers, they are often initialized to a constant value. Other times we found a gadget that does initialize a register to an analyst-controlled value, however, unless that value is a specific constant, a jump is taken in the same gadget, effectively forcing the analyst to initialize the register with that specific value. For firefox.exe, an analyst can create a ROP chain. The gadgets that have to be used are complex, requiring initialization of several gadgets and memory locations to ensure that jumps are not taken. Since PSHAPE cannot

Binary	PSHAPE	rp++	ropper	ROPgadget
firefox.exe$_W$	6,709	6,182	5,445	6,259
iexplore.exe$_W$	928	888	836	888
chrome.exe$_W$	64,372	58,890	52,991	59,969
mshtml.dll$_W$	1,329,705	1,239,403	1,099,466	1,242,616
jfxwebkit.dll$_W$	1,172,718	1,076,350	960,091	1,086,061
chromium$_L$	5,358,283	5,159,712	4,579,388	5,130,856
apache2$_L$	24,164	22,722	18,061	22,875
openssl$_L$	6,978	6,829	5,377	6,845
nginx$_L$	26,314	25,700	21,081	25,245

Table 4.4: Number of gadgets found by each tool on the given binaries, as determined by our evaluation. L denotes Linux and W, Windows.

Function	PSHAPE	ropper	ROPgadget
VirtualProtect$_W$	2/4	n/a	n/a
mprotect$_L$	4/4	1/4	1/4
mmap$_L$	3/3	n/a	n/a

Table 4.5: It is possible to build chains to `mprotect` for all four Linux binaries, line mprotect shows how many of those chains each tool creates. For `mmap`, only three of the Linux binaries have the necessary gadgets to build a chain and this line shows how many of those each tool is able to create. Chains to `VirtualProtect` exist in four out of the five Windows binaries, this line shows how many of them each tool creates. n/a indicates that the tool does not support calling a function that requires the tool to initialize the required number of arguments. L denotes Linux and W, Windows.

yet handle flags, it filters such gadgets out, as it cannot fulfil the preconditions in order to guarantee that the correct branch is taken. Therefore, it was unable to automatically generate a chain in this case.

4.3.6 PSHAPE in Practice

In this chapter, we evaluate three chains that were created fully automatically by PSHAPE. Note that we assume a realistic, unrestricted environment, i.e., PSHAPE can use arbitrary gadgets. Later, in Chapter 5, we evaluate PSHAPE under constraints such as CFI and mitigations monitoring the program to detect ROP-like behaviour.

Chain for apache2. The chain is presented in Figure 4.7. Gadgets 2 to 7 are used to initialize the registers used for passing parameters. After computing permutations of those chains and their pre- and postconditions, PSHAPE detects that `rax` is dereferenced by gadget 6 and before that, aliased with `ebp` (gadget 4). Therefore, another gadget is added that initializes `rbp`, allowing the whole chain to execute correctly. An even shorter chain could have been created by arranging the gadgets in such a way, that gadgets 7 and 4 execute before gadget 6. In this case, gadget 7 initializes `rbp`, gadget 4 copies it to `rax`, which is then dereferenced by gadget 6. This would make the first gadget unnecessary. However, PSHAPE does not detect that, as it uses the first permutation whose postconditions are correct (see Chapter 4.3.3).

G1	`0x3ebe8`	`pop rbp ; ret ;`
G2	`0x46774`	`pop rdi ; ret ;`
G3	`0x57abd`	`pop rsi ; ret ;`
G4	`0x7800d`	`pop rcx ; mov eax, ebp ; add rsp, 8 ;` `pop rbx ; pop rbp ; ret ;`
G5	`0x41200`	`pop rdx ; pop rbx ; ret ;`
G6	`0x4d552`	`pop r8 ; mov rax, qword ptr [rax] ;` `ret ;`
G7	`0x7800c`	`pop r9 ; mov eax, ebp ; add rsp, 8 ;` `pop rbx ; pop rbp ; ret ;`

Figure 4.7: Gadget Chain for apache2

Chain for nginx. The chain is presented in Figure 4.8. In the first iteration, the chain consists of gadgets 3 to 8, which are used to initialize the registers used for passing parameters. Gadget 6 dereferences `rax` and `rbx`, which is why PSHAPE initializes these two registers by adding gadgets 1 and 2 to the chain. Gadget 8 dereferences `rbx`, which is initialized by the seventh to last instructions of gadget 6.

Chain for chrome.exe. Originally, PSHAPE could not create a chain. For exemplary purposes we disabled the filter that removes gadgets containing conditional instructions and present the result in Figure 4.9. Gadgets 2 to 5 initialize the registers used for passing parameters. PSHAPE correctly detected that there are no better-suited gadgets for initializing `r9` and resorts to using gadget 2, prepended by gadget 1 to make `r15` analyst-controlled. As stated before, PSHAPE currently ignores flags, hence it is not able to automatically satisfy the precondition of the `cmovns` instruction, which checks the `sign` flag. To ensure the chain executes correctly, the analyst has to prepend, e.g., a simple `xor rax, rax ; ret` gadget to the chain. Since this chain is incomplete, we do not count it as successful creation.

G1	0x412dab	pop rax ; add rsp, 8 ; ret ;
G2	0x45d594	pop rbx ; ret ;
G3	0x406c20	pop rdi ; ret ;
G4	0x42892b	pop rsi ; ret ;
G5	0x425242	pop rcx ; ret ;
G6	0x444965	pop r8 ; mov qword ptr [rax], rbx ; mov rax, qword ptr [rsp + 8] ; mov qword ptr [rbx + 0x28], rax ; mov rax, qword ptr [rsp + 0x18] ; mov qword ptr [rbx + 0x18], rax ; mov edx, 0 ; mov rax, rdx ; add rsp, 0x58 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret ;
G7	0x45a8c4	pop rdx ; ret ;
G8	0x424219	mov r9, qword ptr [rsp + 0x28] ; mov qword ptr [rbx + 0x48], r9 ; mov r10, qword ptr [rsp + 0x30] ; mov qword ptr [rbx + 0x50], r10 ; mov r11, qword ptr [rsp + 0x38] ; mov qword ptr [rbx + 0x58], r11 ; add rsp, 0x48 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret ;

Figure 4.8: Gadget Chain for nginx

G1	0x56c63	pop r15 ; ret ;
G2	0x25272	cmovns r9d, dword ptr [r15] ; ret 0x2b48 ;
G3	0x9fec6	pop r8 ; ret ;
G4	0x385da	pop rdx ; ret ;
G5	0xa15d3	pop rcx ; ret 0x6e9 ;

Figure 4.9: Gadget Chain for chrome.exe

Note that any padding required between gadgets, e.g., to account for additional `pop` instructions or constants added to `rsp`, is added automatically, but omitted here to increase readability.

4.3.7 Summary

This chapter presented PSHAPE, a tool to partially automate ROP chain generation. To make automation possible, some assumptions are necessary, such as that an attacker wants to invoke a function as quickly as possible. This assumption is strict but realistic. However, even then the number of useful gadgets is likely still very large, and iterating through all possible permutations is not feasible. To reduce the number of gadget candidates further, a metric to assess the quality of a gadget is required. This allows ranking the available gadgets which, in turn, allows PSHAPE to use the most suitable gadgets first. The next chapter introduces such a metric. Once PSHAPE found a valid chain, it automatically adds gadgets to make all dereferenced registers analyst-controlled.

Furthermore, PSHAPE creates summaries, a concise representation of a gadget's semantics. This is very valuable for analysts, as trying to understand what gadgets consisting of 20 or more instructions do, is a difficult task. These summaries can then be used to quickly find gadgets that assist in bypassing other mitigations techniques, for example to find primitives that allow arbitrary reads.

4.4 Gadget Selection

As the previous chapter shows, the search space grows infeasibly large quickly, restricting the approach of simple random combination of all gadgets to very small binaries which have few usable gadgets. A better solution that scales is to pre-select gadgets which have a high probability to work well in practice. To do this, we propose to order gadgets according to their quality, which allows us to target the search towards paths that are more likely to succeed. We achieve this using a tool called GaLity. GaLity is a standalone tool and uses four metrics to assess the quality of a set of gadgets. This allows comparing binaries with respect to their usefulness for ROP. In PSHAPE we utilize one of these metrics, which grades individual gadgets based on various properties, such as effects on registers and memory, side effects, preconditions, and changes to `rsp`. We propose analysts run GaLity on interesting binaries first to get an overview over which binaries will be the most useful ones in terms of gadgets and then, in a second step, apply PSHAPE to them. As we discuss in Chapter 5.4, this can drastically reduce the time it takes to create a ROP chain.

In general, attackers favor simple gadgets that have a minimum of side effects and preconditions. For example, consider a gadget that loads the value that `rsp` points to into `rax`. A clean and effective gadget for achieving this would be: `pop rax ; ret`. In contrast, the gadget: `pop rax ; push rsp ; pop rbp ; mov [rdi+0x34fa], rsp ; ret 0x2dbf1` will also achieve this goal, but will also have the side-effect of overwriting `rbp`. In addition, this gadget has the precondition that `rdi+0x34fa` has to point to writeable

memory. Finally, `ret 0x2dbf1` not only adds a large offset to `rsp` (which can be an issue if attacker-controlled memory is scarce, because it might set `rsp` to point outside of the allocated memory), it also misaligns the stack pointer, which is something normal programs do not do, hinting at a possible exploit execution.

4.4.1 Assessing Gadget Quality

In general, evaluating the quality of a set of gadgets is non-trivial. This originates primarily from the fact that an attacker's goal is potentially unknown, and that given sufficient gadgets, one can construct practically any program. In addition, the gadgets required by an attacker to achieve a goal vary by operating system and architecture. For example, on Windows x86, parameters to functions are usually passed on the stack, while on Windows x86-64 and x64, the first four parameters are passed through registers and all remaining ones are passed on the stack [105], leading to differences in gadget requirements. As a running example, we consider exploits targeting `VirtualProtect`, which is an API call that commonly serves as an avenue to bypassing NX protection on Windows 7 x64 [55, 87, 128]. We stress that our four metrics are not bound to evaluating this specific API call, as they consider the more general attack setup and execution procedures associated with ROP exploits. In addition, we perform an in-depth analysis of the various properties of gadgets with respect to their side effects, preconditions, usability, and usefulness.

Metric 1: Gadget Distribution

The *gadget distribution* metric is calculated by partitioning a given set of gadgets into twelve broad categories, with each category representing a class of operations, such as *arithmetic* and *data move*, as shown in Table 4.6. Gadgets are assigned to a category based on the first instruction of a gadget. For example, the gadget `add rax, 0x40 ; pop rcx ; ret` would be assigned to the *arithmetic* category. We categorize on the basis of the first instruction because every suffix of a gadget is itself a gadget, and will be categorized separately. Note that gadgets containing privileged or sensitive instructions [84] are discarded and not considered in further steps because they trap in user mode, thereby making a gadget unusable.

Analysing the frequency distribution of gadgets amongst categories is helpful as it allows comparing whether the distribution of gadgets in a transformed binary is similar to the one in the original binary, or if the number of gadgets in a category useful for an attacker has grown. Gadget quality and usefulness, however, are not measured and addressed by the remaining metrics.

While Table 4.6 does not contain all instructions of the x86-64 instruction set, it covers 99% of the instructions found in gadgets of the binaries we used in the evaluation, i.e., a

Table 4.6: Gadget Categories

Category	**Included Instructions**
Data move	`pop, push, mov, xchg, lea, cmov, movabs`
Arithmetic	`add, sub, inc, dec, sbb, adc, mul, div, imul, idiv, xor, neg, not`[7]
Logic	`cmp, and, or, test`
Control flow	`call, sysenter, enter, int, jmp, je, jne, jo, jp, js, lcall, ljmp, jg, jge, ja, jae, jb, jbe, jl, jle, jno, jnp, jns, loop, jrcxz`
Shift & Rotate	`shl, shr, sar, sal, ror, rol, rcr, rcl`
Setting flags	`xlatb, std, stc, lahf, cwde, cmc, cld, clc, cdq`
String	`stosd, stosb, scas, salc, sahf, lods, movs`
Floating point	`divps, mulps, movups, movaps, addps, rcpss, sqrtss, maxps, minps, andps, orps, xorps, cmpps, vsubpd, vpsubsb, vmulss, vminsd, ucomiss, subss, subps, subsd, divss, addss, addsd, cvtpi2ps, cvtps2pd, cvtsd2ss, cvtsi2sd, cvtsi2ss, cvtss2sd, mulsd, mulss, fmul, fdiv, fcomp, fadd`
Misc	`wait, set, leave`
MMX	`pxor, movd, movq`
NOP	`nop`
RET	`ret`

total of 20 MiB containing over one million instructions. Due to the large size of the x86-64 instruction set (over 700 instructions [84]), it would be a time-consuming, manual process to cover all existing instructions. However, the fact that we do not achieve 100% coverage does not pose a threat to the metric, because all important and common instructions are categorized. The few we we did not include do not have a big impact on the overall distribution. A manual inspection of uncategorised instructions in other binaries (we used several Windows 7 system libraries) revealed that there were many different instructions but in small numbers in any of the inspected binaries, which is what we expected.

Metric 1, Gadget Distribution, allows to assess whether a transformed binary contains more gadgets in categories useful to an attacker.

Metric 2: Gadget Environment Setup Capabilities

When constructing a ROP chain, an attacker must be able to prepare the environment and operands for subsequent gadgets in a chain. For example, when attempting to perform a Windows API call via ROP, an attacker will generally require the ability to specify the call's arguments. The degree of ease with which an attacker may manipulate memory will affect the choice of gadgets that she uses. In this metric, we consider the most general case, whereby an attacker is able to inject arbitrary arguments into a target program's memory space at a known location. This could be possible due to, e.g., a browser with Javascript turned on, allowing heap sprays and Heap Feng Shui [153], and other vulnerabilities like information leaks [143]. We further assume the vulnerable program is running on a Windows 7 x64 machine, which is a very common platform.

Consider the case whereby an attacker wants to invoke `VirtualProtect`, which takes four arguments. On the aforementioned target platform, the first four parameters are passed through registers (`rcx, rdx, r8, r9`). In such a scenario, an attacker needs to make sure that those registers contain the correct values before `VirtualProtect` can be invoked. To achieve that, three different kinds of gadgets are required, namely: (i) a *stack pivot* gadget which points `rsp` to the injected data, i.e., function arguments and addresses of gadgets, (ii) gadgets to load the arguments from memory to the appropriate registers, and (iii) a gadget that calls `VirtualProtect`.

This metric looks for gadgets that achieve these goals and distinguishes between gadgets that achieve only the required task or include other instructions. Of course, our tool reports gadgets only if the register that receives the argument is preserved, i.e., not overwritten by another instruction in the same gadget. In case the attacker wants to invoke an API that requires fewer arguments, like `VirtualAlloc` [104], fewer gadgets that load arguments are required.

A gadget is only useful in preparing a destination register r_d for use within a ROP chain if it does not destroy its value prior to returning. More concretely, consider a gadget consisting of a sequence of n instructions $i_0; i_1; \ldots i_{n-1};$`ret`. If i_0 assigns the value to r_d, any subsequent instruction i_k with $k > 0$ that has r_d as a target operand and falls within the *data move*, *arithmetic*, or *shift and rotate* categories is tagged as being potentially destructive. A second refinement step is subsequently carried out, whereby the quirks of the target architecture are

[7] It might appear peculiar that `xor, neg, not` are in the arithmetic category - however, this is how exploit developers often use these instructions. Since using nullbytes is sometimes prohibited by the environment, writing the negated or xor-ed value in memory, loading it to a register and then using the same operation on it again is used to bypass this restriction.

taken into account. For instance, instructions that output to a 32 bit subregister are handled differently than those that output to 16 or 8 bit subregisters. This is due to the behaviour that writing to a 32 bit subregister automatically zero-extends the value to fill the entire 64 bit register [84].

In the case of exploits making use of `VirtualProtect`, one finds that three of the four arguments that this API call takes (namely `lpAddress`, the start address of the memory region whose protection level is to be changed, `dsSize`, the size address of the memory region whose protection level is to be changed, and `lpflOldProtect`, an address where the old protection level will be stored) do not need to be precise. If `lpAddress` is a few bytes off an attacker can take this into account, just like a slightly smaller or larger size argument. `lpflOldProtect` is not used by an attacker and can therefore be written to any location. Therefore, the metric only deems two instructions destructive, namely `pop` and `mov` in 64 bit or 32 bit subregisters, as they overwrite the whole register.

Metric 2, Gadget Environment Setup Capabilities, allows one to assess whether a transformed binary contains gadgets typically required for an attack where the environment gives the attacker a lot of leeway.

Metric 3: Gadget Environment Setup Capabilities - Restricted

In contrast to the previous metric, this metric considers the case where an attacker is restricted in the ways in which she can inject values into memory. In particular, we consider the scenario where an attacker may only inject data and hijack the control-flow via `strcpy`. This complicates the direct injection of values into memory because many parameters to API calls often contain null-bytes, which terminate strings, thus requiring that the arguments to be used for correctly invoking a function such as `VirtualProtect` be calculated dynamically at runtime. Imagine an attacker wants to indeed invoke `VirtualProtect`. By taking a look at the required parameters it becomes clear that many will contain null-bytes: `lpAddress` should point to the payload. Depending on the memory layout, this address may contain null-bytes (e.g., in a classic stack buffer overflow vulnerability on Windows, stacks are located at very low addresses making it very likely for the address to have its leftmost bytes set to null). `dwSize` must not be too large, i.e., `lpAddress + dwSize` must include only mapped pages. The value must also not be too small, as it has to cover the memory area where the payload is injected. Typically, the value is a couple of thousand bytes or smaller, which is a value that cannot be injected directly. `flNewProtect` is usually set to `0x40`, which cannot be injected directly because the leftmost bytes are null,and requires to be computed at runtime. `lpflOldProtect` will receive the old protection value,

hence must point to writable memory, which may contain null-bytes. This example shows that in a scenario where the attacker is restricted, she will require various arithmetic and data-move gadgets in order to dynamically calculate parameters for API calls using gadgets.

The metric gauges the presence of gadgets that may be used to assist in evaluating values dynamically at runtime, specifically gadgets that move data between memory and registers and compute values: `pop`, `push`, `add`, `sub`, `adc`, `dec`, `inc`, `neg`, `not`, `mov`, `sbb`, `xchg`, `xor`. As in the case of *Metric 2*, a gadget is only considered if r_d is preserved.

Metric 3, Gadget Environment Setup Capabilities - Restricted, allows one to assess whether a transformed binary contains gadgets typically required for an attack where the attacker has to make many calculations at runtime and cannot inject arbitrary data into a program.

Metric 4: Gadget Quality

The aforementioned metrics do not measure the quality of a gadget per se, rather they provide an indication whether a specific attack can succeed given a set of gadgets. This metric focuses on assessing the quality of an individual gadget, whereby a high-quality gadget is defined as one having no preconditions or side-effects on other registers or memory. An example of a precondition is that a specific register has to point to writeable memory, e.g., in the gadget `pop rax ; mov [rdi+0x34fa], rsp ; ret`. To be usable, `rdi+0x34fa` must point to writeable memory. A side-effect is, for example, that data in another register is overwritten or the stack pointer is manipulated in a way that is difficult to undo, e.g., in the gadget `pop rax ; mov rcx, 0xb0adffff ; leave ; ret`. This gadget overwrites the values in `rcx`, `rsp`, and `rbp`. To express gadget quality, a score is calculated for every gadget considered useful (see *Metric 3*). The score starts at 0 and is increased for side-effects and preconditions. Therefore, a higher score equals worse gadget quality. In the following we give a high-level overview of the two criteria we use to calculate the score for gadget quality.

Scoring Instructions

To measure side-effects and preconditions, the metric inspects every instruction in a gadget. It reuses the categories introduced in Chapter 4.4.1 and assigns a score to each category, which reflects how destructive the instructions in the respective category are. Table 4.7 summarizes the scoring system. Depending on the destination of the instruction, we apply a modifier to the originally assigned score. The metric recognizes three possible kinds of destinations: `rsp`, which should ideally not be modified, because it is responsible for the control

flow and always needs to point to the next gadget. Therefore, modifications of `rsp` usually have the largest influence on the overall score of a gadget. The second possible destination is r_d, the destination register in the first instruction of a gadget, for which we assume that this is also the register an exploit developer is interested in not being modified later on in the same gadget (in case a memory address is the target there is no active register; in case of an `xchg` instruction, both registers are active registers). Modifications of r_d are generally not desirable, but, depending on the modification, can be reversible, e.g., simple arithmetic. The third possible destination is any other general purpose register, except `rsp` and r_d, the metric considers all undesirable side effects and preconditions. Even if they do not affect `rsp` or r_d directly, they still negatively impact the final score.

Table 4.7: Rules for grading instructions. Category describes the category of the instruction (see Table 4.6). "RSP", "rd" and "Other" are possible targets for instructions, the stack pointer, the destination register of the first instruction of a gadget, or any of the other general purpose registers respectively. Categories not in the table generally do not affect the score, with some exceptions discussed in Chapter 4.4.1

Category	**RSP**	**rd**	**Other**	**Notes**
Data move	2	1	0.5	As opposed to all other instructions in this category, `push` does not affect the score of a gadget, since the only side effect it has is on `rsp`, and changes to `rsp` are covered by our `rsp` monitoring.
Arithmetic	2	1	0.5	Arithmetic instructions that modify a register other than `rsp` can be taken into account by the exploit developer. E.g., if `r8` should contain `0x40`, and a gadget like `pop r9 ; add r8, 0x10 ; ret` has to be executed as the last gadget, the developer can simply make sure `r8` contains the value `0x30` before invoking the last gadget. Arithmetic instructions modifying `rsp` are covered by our `rsp` monitoring.
Shift & Rotate	3	2	0.5	These instructions are handled similarly to arithmetic instructions, however, they are more difficult to take into account, which is why they increase the score more than arithmetic instructions.

Because some instructions have side-effects which need to be taken into account, we require a few exceptions in addition to these rules. *Exception #1:* Certain instructions that

modify `rsp` need to be treated differently. This covers all instructions where we can statically determine the offset applied to `rsp`. Depending on how much `rsp` is changed, we adjust the overall score of the gadget. The details on this are covered in the next paragraph. In case it is not possible to statically determine the offset (e.g., `leave` or `pop rsp`), the overall score of the gadget is increased depending on the category of the instruction, as presented in Table 4.7. *Exception #2:* The `leave` instruction does not fall in any of the categories covered by the rules in Table 4.7 but must be graded as it affects `rsp`. This is taken care of through our `rsp` monitoring. *Exception #3:* Remember from Chapter 4.4.1 that we do not cover all of the x86-64 instructions. This means that in very rare cases (less than 0.1%) we cannot grade a gadget because it contains an instruction which we did not categorize. We discard these gadgets from the analysis. *Exception #4:* If an instruction uses a dereferenced register as destination, its score is increased according to the rules in Table 4.7, because this poses a precondition - e.g., the gadget `pop r8 ; mov [rdx], 0xfffa ; ret` has the precondition that `rdx` has to point to writable memory before the gadget can be used.

Monitoring `rsp` Offset

As already alluded to in the previous paragraph, modifications to `rsp` need to be tracked for each gadget. A short example will make clear why this is necessary. Assume the following gadget: `pop rax ; add rbx, 0x10ff ; push rcx ; ret`. In this case, `rsp` will point to the value of `rcx` which was pushed on the stack and jump to it, which is not the injected address of the next gadget. For keeping track of the `rsp` offset the metric uses an SP-Score, *SPS*, which starts at 0, is increased for `pop` and decreased for `push` and `ret` *n* instructions. Of course, also arithmetic instructions on `rsp` are monitored and the respective value is added to or subtracted from *SPS*. When all instructions in a gadget have been analysed and *SPS* is not 0 this means that `rsp` does not point to the next gadget, which might be problematic. A positive score can be thwarted by adding padding, i.e., arbitrary data. A negative score, however, means that `rsp` will point to an address earlier used or data outside the memory area the attacker can control. Therefore, if *SPS* is negative, the overall score of the gadget will be increased by 2. Also, if *SPS* is large (more than 4 KiB) or not a multiple of 4 (for 32 bit binaries) or 8 (for 64 bit binaries), the score of the gadget will be increased by 1, as the former requires an attacker to be able to control more memory and the latter indicates a misaligned stack pointer, which can be detected easily by exploit mitigation tools. If the instruction that operates on `rsp` takes a register and not an immediate (e.g., a `add rsp, rcx`), *SPS* is not changed but the gadget score will be increased by rules in Table 4.7.

Metric 4, Gadget Quality, allows one to assess the overall "quality" of a set of gadgets in respect to side-effects, preconditions, and usability.

4.4.2 Discussion of the Metrics

We believe that metrics that measure the quality of a set of gadgets should focus on practical relevance rather than a theoretical concept such as Turing completeness [144]. Furthermore, they should also reflect whether real-world exploits can be constructed. Since at least Microsoft has seen a shift from classic, stack-based vulnerabilities to heap-related vulnerabilities [10], we believe that metrics should still consider both of these classes of attacks. Last but not least, the metrics should not be limited to well-defined and realistic attack scenarios, but also express overall gadget quality, i.e., side-effects and preconditions. To summarize, metrics as described above should:

- Be practical, i.e., applicable to real scenarios
- Measure if popular current attacks are possible with a given set of gadgets
- Measure if popular past attacks are possible with a given set of gadgets
- Measure gadget "quality"

The proposed metrics achieve all these goals. We would like to stress that our aim is to assess whether a binary contains gadgets suitable for today's ROP attacks. Recently, attacks that use longer and more complex gadgets have been proposed by researchers [26,49,70,71,139]. Such attacks are designed to bypass specific mitigation techniques, which are not yet used in the real world. Thus, in current environments, these complex attacks are cumbersome as they offer no advantage over using regular and simpler ROP gadgets, and we are not aware of any of these complex attacks being used in the wild.

Because of the lack of practical relevance, we decided not to treat gadgets potentially useful in such complex attacks differently than the other gadgets. Nevertheless, if new mitigations that limit the gadgets an attacker may use become widespread and attackers are forced to use more complex and longer gadgets and start using tools that assist in finding gadgets semantically rather than through simple pattern matching, our metrics should be updated to reflect this new environment. This is why we also plan to use a more abstract interpretation of gadgets and look into leveraging synergies created by combining gadgets in the future. Furthermore, we also leave an extension to jump-oriented programming (JOP) [18, 28] for future work.

4.4.3 Evaluation

We have implemented the described metrics in a tool named GaLity, which takes a textfile describing gadgets as input and outputs the metrics we described in Chapter 4.4.1. We demonstrate that it is both practical and useful by applying it to binaries that are compiled to use MPX [132], Intel's latest mitigation technique against runtime exploits. MPX introduces new registers that contain the lower and upper bound of a pointer, and instructions that operate on those registers. This enables compilers to emit additional instructions (MPX and non-MPX) that tracks the sizes of buffers and accesses to those buffers at runtime, which can prevent buffer overflows. On processors which do not support MPX, MPX instructions execute as `nop`, making MPX compatible with older CPUs, but leaving those binaries unprotected by MPX. Given this observation one thus must wonder if the increased code size and thus increased availability of gadgets might actually decrease a binary's security on such systems. We then compare the results obtained by applying GaLity to binaries compiled with MPX support with the results obtained by applying GaLity to the same binaries compiled without MPX support, and determine which binaries, according to our metrics, contain more helpful gadgets for an attacker.

To discover gadgets and write them to a file we used ROPgadget 5.4 [136], with a maximum gadget length of 15 bytes. For this specific case study we decided to consider duplicate gadgets and not just unique gadgets, because if an important gadget exists in a binary several times, this binary is more attractive to an attacker than a binary which contains only one copy of that gadget. This matters, for example, in a scenario where a patch (security-related or not) or any other program modification removes said gadget. Furthermore, taking duplicate gadgets into account helps us measure if the additional gadgets introduced by MPX are copies of useless or useful gadgets.

We compiled programs taken from SPEC CPU2006, using Intel's latest GCC release with MPX support at the time of writing (5.0.0) [158]. We decided to use the SPEC suite because it covers a wide range of application types, and present parts of real programs. MPX is still new and not integrated too well in build chains, which made compiling any program a challenge. However, we got the following eight programs to work properly: 401.bzip2, 403.gcc, 435.gromacs, 456.hmmer, 458.sjeng, 464.h264ref, 473.astar, 482.sphinx3. We compiled all binaries four times, with and without MPX and with and without optimizations (-O2). However, for our evaluation we only considered optimized binaries as this reflects real-world binaries.

First of all, we noticed that MPX has a big influence on file size. With no optimizations, an MPX binary is, on average, almost 3 times as large as a non-MPX binary. With optimization level 2, which we used throughout our experiments, an MPX binary is still, on average, 86% larger compared to a non-MPX binary. We noticed that, while the file size increases by

Table 4.8: Results for *Metrics 2*, *3*, and *4*. Columns rcx, rdx, r8 and r9 denote the number of gadgets which load a value in the respective register, column pivot denotes the number of stack pivot gadgets. The first number denotes the number of gadgets without side-effects, the second number the number of gadgets with side-effects. Column call denotes the number of gadgets usable for indirect calls. These numbers are required for computing *Metric 2*. Column useful denotes the number of useful gadgets, calculated by *Metric 3*. Column Q denotes the number of gadgets with a score of 1 or lower, calculated by *Metric 4*.

	Metric 2						*Metric 3*	*Metric 4*
Program	**rcx**	**rdx**	**r8**	**r9**	**pivot**	**call**	**useful**	**Q**
h264ref	4 / 29	1 / 8	1 / 9	0 / 0	0 / 453	62	6,056	3,749
h264ref MPX	7 / 29	0 / 23	1 / 3	0 / 1	0 / 666	91	7,546	4,906
gromacs	228 / 320	39 / 135	0 / 2	0 / 0	0 / 1071	84	10,823	6,563
gromacs MPX	228 / 418	36 / 141	0 / 7	0 / 1	0 / 1214	155	13,002	8,170
hmmer	6 / 24	3 / 27	0 / 3	0 / 0	0 / 509	33	5,539	3,303
hmmer MPX	8 / 21	4 / 19	0 / 2	0 / 0	0 / 469	39	6,188	3,952
gcc	4 / 71	2 / 219	0 / 14	0 / 8	6 / 5295	588	50,766	32,949
gcc MPX	2 / 52	4 / 71	0 / 9	0 / 4	0 / 4337	763	59,522	39,342
sphinx3	2 / 14	0 / 11	0 / 0	0 / 0	0 / 230	29	3,189	1,964
sphinx3 MPX	1 / 11	0 / 7	0 / 0	0 / 0	1 / 251	52	3,484	2,323
sjeng	1 / 3	0 / 3	0 / 0	0 / 1	0 / 122	72	1,444	983
sjeng MPX	1 / 4	0 / 5	0 / 0	0 / 0	0 / 137	76	1,982	1,414
astar	1 / 4	0 / 4	0 / 0	0 / 0	0 / 122	11	1,009	584
astar MPX	0 / 5	0 / 2	0 / 0	0 / 0	0 / 140	12	1,203	698
bzip2	0 / 1	0 / 1	0 / 0	0 / 0	0 / 99	13	790	466
bzip2 MPX	0 / 1	0 / 1	0 / 0	0 / 0	0 / 112	16	987	605

a factor of almost two, the number of gadgets does not increase in the same way, MPX binaries contain, on average, only 23% more gadgets than non-MPX binaries. This is because the number of gadgets is directly related to the number of `ret` instructions in a binary. MPX does not add many new functions but rather makes existing functions longer, therefore only few intended new `ret` instructions appear. Unintended `ret` instructions [135] might appear in some cases, however, since the new opcodes introduced by MPX do not contain a `ret` opcode, the possibility for this is rather low.

Analysing the increase or decrease of gadgets for each category due to MPX, illustrated in Figure 4.10, shows that most categories gain gadgets. Arithmetic gadgets, which are helpful to an attacker, increase in both number and diversity. Data-move gadgets grow in numbers, but do not change a lot in respect to diversity. An interesting observation is that

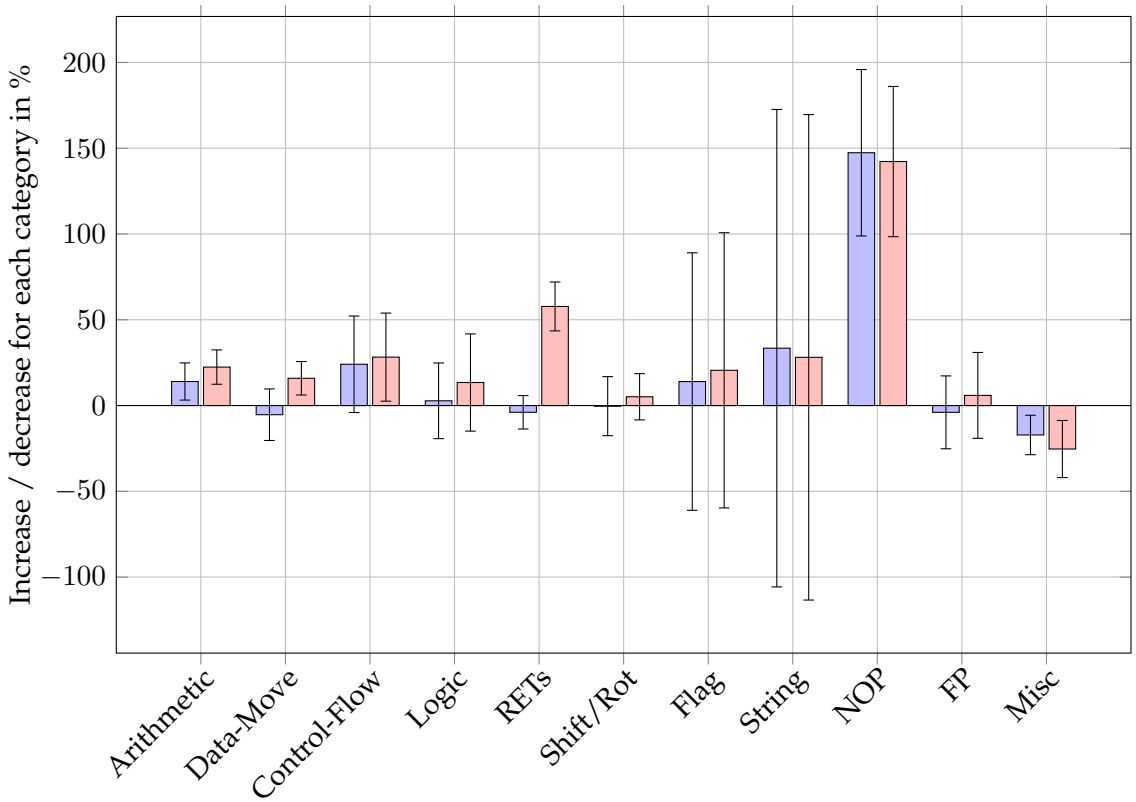


Figure 4.10: This figure shows the average growth of gadgets for each category due to MPX across all eight applications. The blue bar represents the increase considering only unique gadgets, while the red bar represents the total increase of gadgets, i.e., also duplicate gadgets. We use the information about how the number of unique gadgets changes to infer if and how gadget variety is affected by a program transformation.

NOP-gadgets increase drastically, which is presumably due to the fact that the new MPX instructions are interpreted as multi-byte `NOP`s on hardware that does not support MPX. The categories flag, string and floating-point have a high standard deviation, indicating that changes in these categories are very application-specific. Gadgets in the miscellaneous category decrease both in diversity and number. Despite the large increase of `nop` gadgets, the overall distribution of gadgets remains roughly the same, as Figure 4.11 shows. Overall we conclude that MPX binaries contain more gadgets in categories helpful to an attacker.

Next, we are interested in the two attack scenarios, i.e., *Metrics 2* and *3*. Regarding *Metric 2*, there is no big difference in the availability of gadgets. Gadgets that load arguments in `r8` or `r9` are rare in both MPX and non-MPX binaries, and sometimes the MPX binary and sometimes the non-MPX binary contains some. Regarding *Metric 3*, the number of useful gadgets increases in every binary and on average by 17%, making MPX binaries a much more attractive target to attackers. We summarize the results in Table 4.8. Lastly, we determine overall gadget quality using *Metric 4*. In all eight binaries, the MPX versions contain

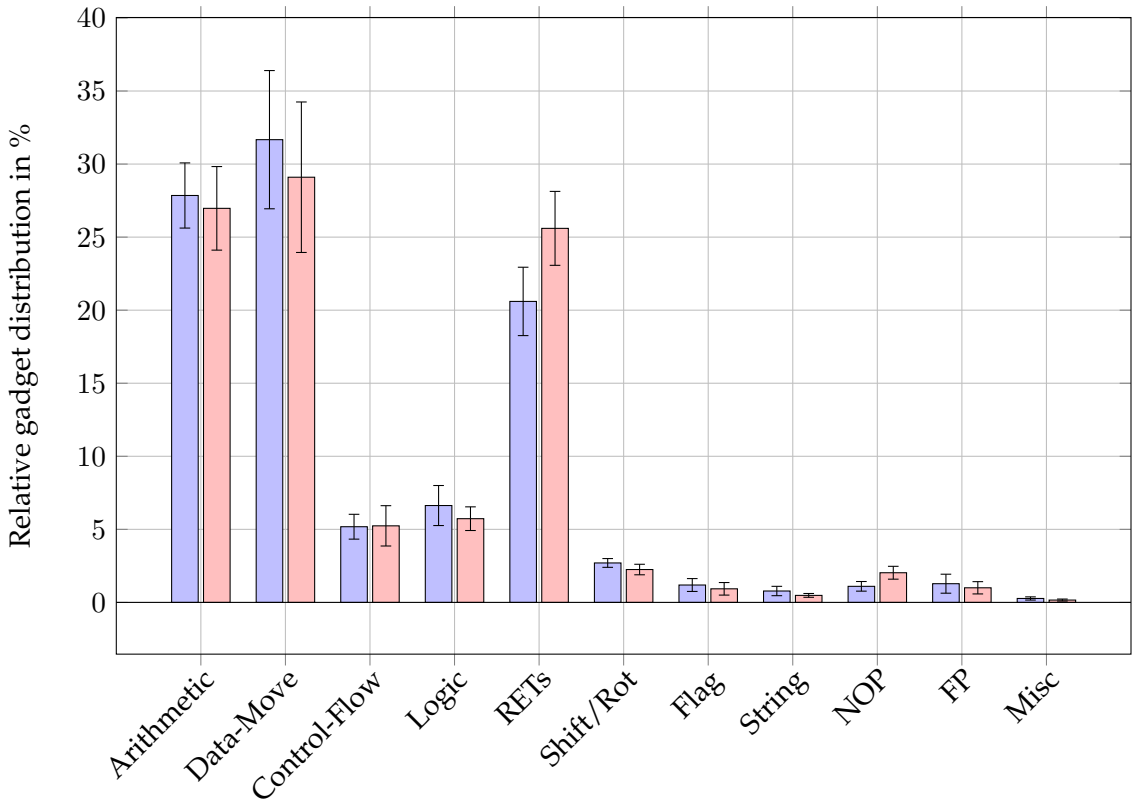


Figure 4.11: This figure shows the average distribution of gadgets across all eight applications. The blue bar represents the non-MPX binaries, while the red bar represents the MPX binaries.

more gadgets of high quality, i.e., with fewer side-effects and preconditions, as the last column of Table 4.8 shows.

By taking all four results into consideration we come to the conclusion, that binaries compiled with MPX support are favourable for an attacker. *Metric 1* shows an overall increase of gadgets in useful categories, further confirmed by *Metric 3*, which also shows that the additional gadgets in those categories are useful in practice. *Metric 2* gives no indication that MPX or non-MPX binaries contain more of the required gadgets. *Metric 4* gives the indication that MPX binaries tend to have more gadgets of higher quality, making them easier to use for an attacker.

4.4.4 Summary

GaLity allows analysts to determine the quality of a gadget or a set of gadgets. This is very important for PSHAPE, because attempting ROP chain automation with hundreds of thousands of gadget candidates is not feasible. The ability to grade gadgets allows PSHAPE to try the gadgets with the highest probability of working first, instead of combining random gadgets. It also allows an exploit developer to quickly assess whether a binary contains a set of gadgets preferable over another binary's gadgets. This is especially important because the process of extracting, analysing, and summarizing gadgets takes a lot of time, as we show in Chapter 5.4. When the exploit developer knows which binaries contain the most useful gadgets, she can apply PSHAPE to those first. We quantify this potential for reducing the time it takes to build a ROP chain in Chapter 5.4.

4.5 Summary

In this chapter we showed how ROP is used in practice. Exploit developers want to execute regular shellcode as soon as possible, because encoding a whole payload using only ROP is cumbersome and overly complicated. To achieve this, an API that changes the protection level of a range of memory is usually invoked. This is a straight-forward task where the attacker has a lot of freedom, because there are no restrictions on what kinds of gadgets an exploit developer can use.

First, to mitigate this issue, we presented AntiCRA, which is a part of ROPocop, a tool that places certain restrictions on the attacker and the ROP chain. ROPocop is a dynamic ROP detector which monitors program execution and picks up on program behaviour that can be linked to ROP or other code-reuse attacks. It is based on a heuristic that is triggered by consecutive indirect control-flow transfers and measures the average length of previous basic blocks. If a certain threshold, that can be set by the user to program-specific levels, is undershot, an alarm is raised.

Then we introduced PSHAPE, a framework to support exploit developers with the creation of ROP chains. It providing helpful, concise summaries for gadgets, which help the developer to determine the effects of a gadget on the program state. It also automatically chains gadgets together, ensuring preconditions are satisfied. PSHAPE battles the usual problems of automation by reducing the search space in such a way, that only the most useful gadgets are considered, greatly reducing the number of gadget candidates.

Lastly, we described GaLity, a set of metrics that determines quality and usefulness of a gadget or a set of gadgets. PSHAPE uses these results to select the most useful gadgets, which enables feasible automation.

Chapter 5

Cross System Case Studies

We evaluated our tools throughout this thesis: in Chapter 4.2.4 we evaluated ROPocop and showed that it is very effective at preventing code-reuse attacks. In Chapter 4.3.6 we evaluated PSHAPE, showing ROP chains it created fully automatically for current, realistic scenarios. In these scenarios, no restrictions are placed on ROP chains, which reflects the current situation. Lastly, in Chapter 4.4.3, we applied GaLity to various binaries and showed its usefulness in assessing gadget quality.

In this chapter, we evaluate how well PSHAPE works against ROPocop, and, more general, how well it works in settings where mitigation techniques restrict the gadgets which are available to the exploit developer. For this we select a subset of the applications used in Chapter 4.3.6, using only applications that PSHAPE successfully created a chain for. Our reasoning for this that if PSHAPE was not able to build a chain before, where no restrictions were in place, either due to its limitations or because necessary gadgets were not available, it will not be able to succeed now, under heavy restrictions. To increase the challenge further, we select the two smallest binaries, which are apache2 and nginx, because smaller binaries tend to contain fewer gadgets. We use three different attacks against two types of mitigations. More precisely we show how PSHAPE

- bypasses heuristics such as AntiCRA, which monitor gadget length, by using only gadgets of a specified minimum length in order not to trigger the heuristic (Chapter 5.1).

- bypasses heuristics such as AntiCRA, which monitor gadget length, by using a heuristic breaker. To achieve this, we extended PSHAPE to find heuristic breaker candidates (Chapter 5.2).

- bypasses mitigations which enforce that every `ret` instruction has to return to after a `call` instruction, i.e., enforces that all gadgets are call-preceded (e.g., [1,120,174,175]).

To achieve this, we extended PSHAPE to filter out gadgets that are not call-preceded (Chapter 5.3).

While the fact that such mitigations can be bypassed is not new [25,26,49,71], our tool is the first one to automate this process.

5.1 Case Study 1

In this scenario, we investigate the creation of a ROP chain that evades heuristics by avoiding the "typical" ROP gadgets, i.e., gadgets a human analyst opts for. Such gadgets are short, achieve a simple task, and have no preconditions or side effects. Instead of using those gadgets, we force PSHAPE to use longer gadgets. Based on the results from our ROPocop evaluation (see Table 4.2), we chose a minimum length of five instructions, which leads to ROP chains with an average gadget length of 5. This is high enough to bypass even heuristics with much stricter thresholds than ours.

Figure 5.1 shows a chain PSHAPE created for Apache 2, initializing all six registers used for passing parameters. Most of them are quite simple, without any aliasing or complex instructions. Gadget 1 initializes `rax`, as it is later dereferenced by gadget 3. It also adds a small constant (`5bh`) to `rax`, so the exploit developer has to keep this in mind when injecting the value that will be loaded into `rax`. Gadget 2 initializes `rdi` and `r8` and overwrites `rbx`, `rbp`, `r12`, and `r13`. This is a common pattern also shown by subsequent gadgets. The reason is that registers `rbx`, `rbp`, and `r12-15` must be preserved by the callee, if it uses them. Therefore, as part of the function prologue, those registers are saved on the stack, then used by the callee during its execution, and restored upon returning to the caller, as part of the function epilogue. Gadget 3 initializes `rsi` and dereferences `rax`, which gadget 1 initialized. Gadgets 4, 5, and 6 initialize `rcx`, `rdx`, and `r9`, respectively, without any adverse side-effects.

Figure 5.2 shows a chain PSHAPE created for nginx, initializing all six registers used for passing parameters. As one can see, the chain is very complex and would have been difficult to create manually. Gadget 1 initializes `rbx`, because it is later dereferenced by gadgets 7 and 8. Gadget 2 initializes `rax`, because it is later dereferenced by gadgets 2, 3, 4, 5, 6, and 8. Note that while `rax` is changed slightly by gadgets 3, 4, and 5, the changes only affect its lower two bytes, which does not cause problems. Gadgets 3, 4, 5, and 6 initialize `rdi`, `rsi`, `rcx`, and `rdx`, respectively. They also dereference `rax`, which has been initialized by gadget 2. Gadget 7 initializes `r9` and dereferences `rbx`, which has been initialized by gadget 1. It also initializes `rbx` again and it is important that the exploit developer makes sure it is initialized with a legal address, as it is dereferenced again by gadget 8. Gadget 8 also initializes `r8` and dereferences `rbx`, initialized by gadget 7, and `rax`, initialized by gadget 2.

G1	`0x4162f`	`pop rax ; add al, 0x5b ; pop rbp ; pop r12 ; ret ;`
G2	`0x781c4`	`pop rdi ; pop r8 ; add rsp, 0x18 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; ret ;`
G3	`0xe7c1`	`pop rsi ; add byte ptr [rax], al ; adc dword ptr [rax], eax ; sbb byte ptr [rax], al ; ret 0x29 ;`
G4	`0x7800d`	`pop rcx ; mov eax, ebp ; add rsp, 8 ; pop rbx ; pop rbp ; ret ;`
G5	`0x41b13`	`pop rdx ; pop rbx ; pop rbp ; pop r12 ; ret ;`
G6	`0x7800c`	`pop r9 ; mov eax, ebp ; add rsp, 8 ; pop rbx ; pop rbp ; ret ;`

Figure 5.1: Gadget chain with long gadgets for Apache 2.

5.2 Case Study 2

In this scenario we bypass heuristics, but, as opposed to the previous scenario, by mixing simple, short gadgets with very long gadgets, thereby artificially increasing the average length of ROP gadgets. This allows us to use mostly simple gadgets, but still bypass heuristics. We achieve this by incorporating heuristic breakers [26, 49, 71] (see Chapter 2.4.3), into the gadget chains. We extended PSHAPE to assist in finding such gadgets. Using our summaries PSHAPE calculates a score for each gadget that fulfils the following two requirements:

- The gadget does not change `rsp` by more than x bytes. Changes to `rsp` are generally undesired, as they require the exploit developer to control a larger memory area. x can be set by the user, in our experiments we used 100 bytes.
- The Gadget contains at least y instructions. y can be set by the user, in our experiments we used 25 instructions.

PSHAPE then calculates a score for each gadget that fulfils these requirements as follows:

$$score = len - (preconds \times 1.2) - (postconds \times 0.8)$$

$preconds$ is the number of registers the exploit developer needs to control because they are dereferenced by the heuristic breaker. $postconds$ is the number of registers the heuristic breaker affects. We weigh these properties slightly differently, because using a heuristic

G1	0x40be80	pop rbx ; mov eax, edx ; shr eax, 0xf ; xor eax, edx ; ret ;
G2	0x45f6a6	pop rax ; pop rbp ; cli ; dec dword ptr [rax - 0x77] ; ret 0xf4b8 ;
G3	0x422383	adc al, ch ; pop rdi ; xor bh, dh ; dec dword ptr [rax - 0x77] ; ret ;
G4	0x4346ba	add al, ch ; pop rsi ; or al, 0xfd ; dec dword ptr [rax - 0x77] ; ret ;
G5	0x403572	add al, ch ; pop rcx ; adc al, byte ptr [rax] ; add byte ptr [rax - 0x39], cl ; ret 0xffff ;
G6	0x41a68a	add byte ptr [rax], al ; add byte ptr [rax - 0x77], cl ; pop rdx ; xor byte ptr [rax - 0x77], cl ; ret ;
G7	0x424219	mov r9, qword ptr [rsp + 0x28] ; mov qword ptr [rbx + 0x48], r9 ; mov r10, qword ptr [rsp + 0x30] ; mov qword ptr [rbx + 0x50], r10 ; mov r11, qword ptr [rsp + 0x38] ; mov qword ptr [rbx + 0x58], r11 ; add rsp, 0x48 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret ;
G8	0x444965	pop r8 ; mov qword ptr [rax], rbx ; mov rax, qword ptr [rsp + 8] ; mov qword ptr [rbx + 0x28], rax ; mov rax, qword ptr [rsp + 0x18] ; mov qword ptr [rbx + 0x18], rax ; mov edx, 0 ; mov rax, rdx ; add rsp, 0x58 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret ;

Figure 5.2: Gadget chain with long gadgets for nginx.

breaker as the first gadget can increase the average enough to safely execute short gadgets afterwards. At the same time, registers will not contain any important, i.e., analyst-controlled data yet, so any effects such a gadget has on registers are of no consequence. However, at the beginning of the chain, the attacker will not have control over many registers, therefore fewer preconditions are preferred. Of course, all constants can be changed by the user, and

different scenarios might require different weighing. However, the settings described above worked very well in our experiments.

Finding heuristic breakers is not a problem in any of the binaries we investigated. Listing 5.1 and Listing 5.2 show a good and a bad example of heuristic breakers. While similar in length (29 and 35 instructions), they greatly differ in regards to their pre- and postconditions. The first heuristic breaker requires the attacker to control six registers (`rax`, `rsi`, `rcx`, `rdi`, `rbp`, and `rbx`) and changes `rax` and `rdx`. On the other hand, the second heuristic breaker requires the attacker to control just one register (`rax`) and changes only the `rflags` register. Therefore, this heuristic breaker can easily be inserted at arbitrary points in a ROP chain, if a sliding window is used to determine average gadget length. Consequently, PSHAPE grades the two heuristic breakers very differently and assigns the better score to the second one.

PSHAPE makes it very easy to quickly find similar heuristic breakers in all binaries. Using any of these gadgets, the average can be increased drastically making them very useful against heuristic-based mitigations. Davi et al. describe how finding such gadgets "was a non-trivial task that required painstaking analyses and a stroke of luck" [49]. The best gadget they found consists of 23 instructions, requiring `esi` and `edi` to point to writeable memory, and overwriting `esi`, `edi`, `eax`, `ebx`, and `ebp`. Other researchers are more vague and mention gadgets of length 33 [71] or 22 [26], however, without giving guarantees about register state. They also do not mention whether they find the gadgets manually or automatically.

We also looked for a "universal" heuristic breaker, i.e., a heuristic breaker that is likely to be available in many environments. For this scenario we set the minimum length to 25 and the maximum to 50 and applied PSHAPE to some core Windows libraries. PSHAPE found one such universal heuristic breaker, shown in Figure 5.3. It is located in `ntdll.dll`, hence will be available in any Windows application. It requires the attacker to control two registers (`rdx` and `rcx`) and changes only `rax`. Depending on the version of Windows and installed patches this particular gadget might not be available, though. However, the same library contains plenty of other, similarly well-suited heuristic breakers.

Assuming an exploit developer wants to keep the average length of the ROP chain above 5, and the gadgets she uses always consist of two instructions, approximately every 16th gadget needs to be a heuristic breaker of length 50. If some gadgets are longer, heuristic breakers are required less frequently.

This case study also shows how easy it is to extend PSHAPE with new functionality. Based on our summaries, new features can often be programmed in easily. Expressing a similar set of rules for finding heuristic breakers directly on assembler code, which is what other tools output, is a lot more difficult. We discuss the topic of extensibility further in Chapter 6.

Listing 5.1: Bad heuristic breaker

```
add   [rax], al
add   [rax - 0x77], cl
test  [rax + rsi*4], ah
add   [rax], al
add   [rcx - 0x75], cl
test  [rax + rdi*4], ah
add   [rax], al
add   [rax - 0x77], cl
test  [rax + rbp*4], ah
add   [rax], al
add   [rcx - 0x75], cl
test  [rax + rcx*2], ah
add   al, [rax]
add   [rax - 0x77], cl
test  [rax + rcx*8], ah
add   [rax], al
add   [rcx - 0x75], cl
test  [rax + rdx*2], ah
add   al, [rax]
add   [rax - 0x77], cl
test  [rax + rdx*8], ah
add   [rax], al
add   [rcx + 0xf], al
mov   dh, 0x84
and   al, 0x58
add   al, [rax]
add   [rbx - 0x1f3efe20], al
add   eax, 0x8908c883
ret   0x41
```

Listing 5.2: Good heuristic breaker

```
nop
nop
add   [rax], al
add   [rax - 0x77], cl
nop
test  al, 0
add   [rax], al
mov   [rax + 0xe8], rdx
mov   [rax + 8], rdx
mov   [rax + 0x10], rdx
mov   [rax + 0x18], rdx
mov   [rax + 0x28], rdx
mov   [rax + 0x30], rdx
mov   [rax + 0x50], rdx
mov   [rax + 0x58], rdx
mov   [rax + 0x60], rdx
mov   [rax + 0x98], rdx
mov   [rax + 0xa0], rdx
mov   [rax + 0xf0], rdx
mov   [rax + 0xf8], rdx
mov   [rax + 0x128], rdx
mov   [rax + 0x130], rdx
mov   [rax + 0x120], rdx
mov   [rax + 0x110], rdx
mov   [rax + 0x118], rdx
mov   [rax + 0xd8], rdx
mov   [rax + 0xe0], rdx
mov   [rax + 0xb0], rdx
mov   [rax + 0xb8], 0
mov   [rax + 0x190], rdx
mov   [rax + 0x1c0], rdx
mov   [rax + 0x150], 7
mov   [rax + 0x158], 0x46dd77
add   rsp, 8
ret
```

```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov    eax, [rdx]
mov    [rcx], eax
mov    eax, [rdx + 4]
mov    [rcx + 0x54], eax
mov    eax, [rdx + 8]
mov    [rcx + 0xa8], eax
movzx  eax, [rdx + 0xc]
mov    [rcx + 0x44], ax
movzx  eax, [rdx + 0xe]
mov    [rcx + 0x46], ax
movzx  eax, [rdx + 0x10]
mov    [rcx + 0x52], ax
movzx  eax, [rdx + 0x12]
mov    [rcx + 0x48], ax
...
```

```
...
movzx  eax, [rdx + 0x14]
mov    [rcx + 0x4a], ax
movzx  eax, [rdx + 0x16]
mov    [rcx + 0x4c], ax
movzx  eax, [rdx + 0x18]
mov    [rcx + 0x4e], ax
movzx  eax, [rdx + 0x1a]
mov    [rcx + 0x50], ax
movzx  eax, [rdx + 0x1c]
mov    [rcx + 0x98], ax
movzx  eax, [rdx + 0x1e]
mov    [rcx + 0x9a], ax
movzx  eax, [rdx + 0x20]
mov    [rcx + 0xa6], ax
movzx  eax, [rdx + 0x22]
mov    [rcx + 0x9c], ax
movzx  eax, [rdx + 0x24]
mov    [rcx + 0x9e], ax
movzx  eax, [rdx + 0x26]
mov    [rcx + 0xa0], ax
movzx  eax, [rdx + 0x28]
mov    [rcx + 0xa2], ax
movzx  eax, [rdx + 0x2a]
mov    [rcx + 0xa4], ax
ret
```

Figure 5.3: Universal heuristic breaker in `ntdll.dll`.

5.3 Case Study 3

Finally, in our last case study, we use PSHAPE to bypass mitigations that restrict the set of gadgets to only call-preceded gadgets, like many CFI implementations do [1, 120, 174, 175]. This drastically reduces the number of gadgets, as Figure 5.4 shows[1]. Usable means without

[1]Carlini and Wagner [26] found that about 6% of gadgets are call-preceded, while according to our experiments, it is only about 3.3%

instructions PSHAPE cannot handle, i.e., no calls, jumps, etc. Therefore, these are gadgets PSHAPE can use for automatically constructing gadget chains. However, keep in mind, that we only look at one isolated binary. Even small programs usually load several OS libraries, and a varying number of program libraries. For larger programs, such as browsers, it is common to have tens of MiB of code to use for ROP.

Binary	Usable Gadgets	Call-preceded Gadgets	Usable call-preceded Gadgets
firefox.exe$_W$	6,709	834	202
iexplore.exe$_W$	928	103	20
chrome.exe$_W$	64,372	7,942	2,271
mshtml.dll$_W$	1,329,705	206,097	60,757
jfxwebkit.dll$_W$	1,172,718	150,456	46,730
chromium$_L$	5,358,283	751,190	189,334
apache2$_L$	24,164	2,141	395
openssl$_L$	6,978	1,022	158
nginx$_L$	26,314	2,046	462

Figure 5.4: Call-preceded gadgets

Figure 5.5 shows a chain PSHAPE generated for nginx. Thanks to a very useful gadget in nginx, it is possible to initialize all registers using just one gadget, which also happens to be call-preceded. Of course, this gadget could also have been used in the evaluation of PSHAPE in Chapter 4.3.6, however, GaLity assigned this gadget a bad score because it is very long and has lots of side effects. We discuss this behaviour in Chapter 5.5. Figure 5.6 shows the original output of gdb for this gadget.

Figure 5.7 shows a chain PSHAPE created for Apache 2, which initializes all six registers used for passing parameters using only call-preceded gadgets. The gadgets are very simple and contain no register dereferences, other than `rsp`, making them very easy to use. Gadget 1 initializes `rdi` and `r8`. Gadget 2 initializes `rcx` and `rsi` and, like Gadget 1, overwrites some callee-saved registers. Gadget 3 initializes `rdx`. Lastly, gadget 4 initializes `r8` again, but, more importantly, `r9`.

5.4 Performance

5.4.1 Gadget Summaries

Keep in mind that PSHAPE does a lot of work that goes beyond simple gadget discovery. In addition to finding and extracting gadgets, it lifts gadgets to VEX IR, analyses them, and

G1	0x40f4c7	pop rbx ; ret ;
G2	0x4241de	mov qword ptr [rbx], r15 ; mov qword ptr [rbx + 8], r14 ; mov qword ptr [rbx + 0x10], r13 ; mov qword ptr [rbx + 0x18], r12 ; mov rcx, qword ptr [rsp] ; mov qword ptr [rbx + 0x20], rcx ; mov rsi, qword ptr [rsp + 8] ; mov qword ptr [rbx + 0x28], rsi ; mov rdx, qword ptr [rsp + 0x10] ; mov qword ptr [rbx + 0x30], rdx ; mov rdi, qword ptr [rsp + 0x18] ; mov qword ptr [rbx + 0x38], rdi ; mov r8, qword ptr [rsp + 0x20] ; mov qword ptr [rbx + 0x40], r8 ; mov r9, qword ptr [rsp + 0x28] ; mov qword ptr [rbx + 0x48], r9 ; mov r10, qword ptr [rsp + 0x30] ; mov qword ptr [rbx + 0x50], r10 ; mov r11, qword ptr [rsp + 0x38] ; mov qword ptr [rbx + 0x58], r11 ; add rsp, 0x48 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret ;

Figure 5.5: Gadget Chain for nginx setting up six parameters using only call-preceded gadgets.

creates semantic summaries. We ran our experiments with a maximum gadget length of 35 on a server with 40 Intel Xeon E5-4640 CPUs, clocked at 2.4 Ghz each, and 224 GB of RAM.

Extraction and analysis of all gadgets up to a maximum length of 35 instructions in our sample set (approximately 147 MiB of code in 9 real-world binaries) takes about 31 hours. This leads to an average throughput of about 4.7 MiB/hour, or 1.3 KiB/s. However, we are dealing with a lot of data: over all binaries PSHAPE finds about 33.5 million gadgets. This leads to a gadget density of roughly 220,000 gadgets/MiB or 215 gadgets/KiB. Hence PSHAPE extracts, lifts, analyses, and summarizes an average of 290 gadgets per second using 40 threads.

PSHAPE periodically dumps its results to the disk, therefore, RAM use is relatively low (about 800 MiB). This allows PSHAPE to run on regular, consumer-grade hardware. While it might take about a day to analyse a binary of 10 MiB using a quad-core CPU, it is certainly possible. Other tools we have evaluated are not able to achieve this in a similar manner.

```
(gdb) x/50i 0x4241d9
   0x4241d9 <ngx_http_core_types+181>:  call   0x411717 <ngx_conf_parse>
   0x4241de <ngx_http_core_types+186>:  mov    QWORD PTR [rbx],r15
   0x4241e1 <ngx_http_core_types+189>:  mov    QWORD PTR [rbx+0x8],r14
   0x4241e5 <ngx_http_core_types+193>:  mov    QWORD PTR [rbx+0x10],r13
   0x4241e9 <ngx_http_core_types+197>:  mov    QWORD PTR [rbx+0x18],r12
   0x4241ed <ngx_http_core_types+201>:  mov    rcx,QWORD PTR [rsp]
   0x4241f1 <ngx_http_core_types+205>:  mov    QWORD PTR [rbx+0x20],rcx
   0x4241f5 <ngx_http_core_types+209>:  mov    rsi,QWORD PTR [rsp+0x8]
   0x4241fa <ngx_http_core_types+214>:  mov    QWORD PTR [rbx+0x28],rsi
   0x4241fe <ngx_http_core_types+218>:  mov    rdx,QWORD PTR [rsp+0x10]
   0x424203 <ngx_http_core_types+223>:  mov    QWORD PTR [rbx+0x30],rdx
   0x424207 <ngx_http_core_types+227>:  mov    rdi,QWORD PTR [rsp+0x18]
   0x42420c <ngx_http_core_types+232>:  mov    QWORD PTR [rbx+0x38],rdi
   0x424210 <ngx_http_core_types+236>:  mov    r8,QWORD PTR [rsp+0x20]
   0x424215 <ngx_http_core_types+241>:  mov    QWORD PTR [rbx+0x40],r8
   0x424219 <ngx_http_core_types+245>:  mov    r9,QWORD PTR [rsp+0x28]
   0x42421e <ngx_http_core_types+250>:  mov    QWORD PTR [rbx+0x48],r9
   0x424222 <ngx_http_core_types+254>:  mov    r10,QWORD PTR [rsp+0x30]
   0x424227 <ngx_http_core_types+259>:  mov    QWORD PTR [rbx+0x50],r10
   0x42422b <ngx_http_core_types+263>:  mov    r11,QWORD PTR [rsp+0x38]
   0x424230 <ngx_http_core_types+268>:  mov    QWORD PTR [rbx+0x58],r11
   0x424234 <ngx_http_core_types+272>:  add    rsp,0x48
   0x424238 <ngx_http_core_types+276>:  pop    rbx
   0x424239 <ngx_http_core_types+277>:  pop    rbp
   0x42423a <ngx_http_core_types+278>:  pop    r12
   0x42423c <ngx_http_core_types+280>:  pop    r13
   0x42423e <ngx_http_core_types+282>:  pop    r14
   0x424240 <ngx_http_core_types+284>:  pop    r15
   0x424242 <ngx_http_core_types+286>:  ret
```

Figure 5.6: gdb output of gadget shown in Figure 5.5

.

G1	`0x781c4`	`pop rdi ; pop r8 ; add rsp, 0x18 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; ret ;`
G2	`0x7820f`	`pop rcx ; pop rsi ; add rsp, 0x18 ; pop rbx ; pop rbp ; pop r12 ; pop r13 ; ret ;`
G3	`0x58c9c`	`pop rax ; pop rdx ; add rsp, 8 ; pop rbx ; pop rbp ; ret ;`
G4	`0x7800a`	`pop r8 ; pop r9 ; mov eax, ebp ; add rsp, 8 ; pop rbx ; pop rbp ; ret ;`

Figure 5.7: Gadget Chain for Apache 2 setting up six parameters using only call-preceded gadgets.

Furthermore, the step of extracting and analysing gadgets has to be done only once, because results generated by PSHAPE can be stored and loaded from the disk.

Our current prototype is not optimized and analyses all gadgets. We want to offer summaries even for gadgets that PSHAPE cannot use (e.g., jumps) or that would, under normal

circumstances, crash the program (e.g., due to privileged instructions). This is probably not required in most cases, and would allow us to, for example, filter out gadgets that contain certain instructions before lifting. As our evaluation of PSHAPE in Table 4.4 in Chapter 4.3.5 shows, after filtering unusable gadget a total of 8 million gadgets remains, i.e., about a quarter. In small-scale experiments this translates directly into performance, speeding up the process by a factor of 4x.

5.4.2 Gadget Chaining

Creation of gadget chains consists of permuting the gadget candidates and then analysing whether a generated chain is valid. Creating permutations takes less than a second, which is why we concentrate on the analysis of the resulting chains in this chapter.

In essence, a gadget chain is simply a series of gadgets. After removing the `ret` instructions, a gadget chain is basically just one long gadget and can be treated as such. Therefore, the analysis of gadget chains in PSHAPE uses the same algorithms it uses for gadget summaries and is therefore comparable in performance. However, since the number of gadgets to analyse is much smaller, it takes less time to run. Assume the same scenario as in Chapter 4.3.5, i.e., creating chains to functions that take four / six parameters and using only the most suitable gadget. This results in 24 / 720 chains or gadgets that PSHAPE needs to analyse. While those gadget chains are more complex than individual gadgets, PSHAPE is still able to do analyse hundreds of chains in a matter of seconds.

5.4.3 Time Savings due to GaLity

We now describe how running GaLity before PSHAPE to grade binaries has the potential to save a large amount of time. GaLity analyses about 17,000 gadgets/s or roughly one million gadgets per minute using only one single thread. Considering a gadget density of 220,000 gadgets/MiB, this translates to almost 5 MiB/min. Let us assume an arbitrary program consists of 20 binaries of varying sizes with a total of 40 MiB of code. Without GaLity, PSHAPE has to be applied to every binary until it succeeds. In the best case, the first binary contains all necessary gadgets. In this case, running GaLity provides no benefit and increases runtime by about eight minutes. On the other hand, in the worst case, where the last binary contains all necessary gadgets, PSHAPE takes about eight hours to finish[2]. This worst case scenario, however, can be prevented by running GaLity first. Unfortunately, it is not easily possible to quantify how precise GaLity's recommendations are, as this would require us to manually assess millions of gadgets, which is not feasible. However, our case study in Chapter 4.4.3 showed that GaLity produces useful results in realistic scenarios.

[2]Keep in mind that this number is based on our original benchmarks, using a server with 40 CPUs.

Furthermore, even in the unlikely case that GaLity completely fails and suggests using the binary which contains the necessary gadgets last, the total runtime increases by only eight minutes, which is negligible considering PSHAPE's runtime.

5.5 Discussion

In Chapter 4.3.6, where we evaluated PSHAPE, we used current, realistic scenarios. In this chapter, we took into account mitigation techniques that are not widely deployed yet and really stress the algorithms PSHAPE uses. Our three case studies showed that PSHAPE is able to automatically generate ROP chains capable of bypassing real mitigation techniques. By using gadgets a human exploit developer or analyst would shy away from and avoid due to their length and complexity, PSHAPE is able to create ROP chains which initialize registers used for passing parameters to functions. While it is not impossible to do this manually, PSHAPE saves a lot of time the exploit developer can spend on other tasks, such as bypassing mitigation techniques. Even in cases where it does not succeed, the summaries it provides are still very helpful in finding useful gadgets.

In case studies 1 and 2, we found that using heuristic breakers are the preferred way of bypassing heuristics-based mitigations. With PSHAPE they are easy to find and present in many binaries. Relying on only very long gadgets on the other hand is more challenging because usable long gadgets may not always exist. The universal heuristic breaker we found in `ntdll.dll` can be used with ease in all ROP chains targeting Windows, and ensures the average length of the ROP chain is high enough to evade even very strict thresholds. The ability to find suitable heuristic breakers is especially important, if pure ROP exploits are used. Short, two-staged exploits using only five gadgets might go unnoticed, however, pure ROP exploits contain hundreds or thousands of gadgets [95]. These gadgets might trigger heuristics, therefore the exploit developer has to incorporate heuristic breakers to artificially increase the average in certain intervals. Case study 2 also shows the importance of our summaries. They allowed us to easily extend PSHAPE and enabled it to find heuristic breakers, which would otherwise have been a complicated process.

In case studies 2 and 3 we came across gadgets that load several or even all registers used for passing parameters. PSHAPE does not use these under normal circumstances, because, due to their length and side effects, GaLity assigns them a bad score. It is debatable whether, when creating a ROP chain, using one or two gadgets that load all registers are preferable over using several short and simple gadgets. An advantage of using just one gadget is that the chance of being detected by a heuristic is very low. On the other hand, such gadgets may have preconditions, simple gadgets often do not have. We believe that in practice both options are equal. In some scenarios, one might be advantageous, for example, when

mitigations prevent either approach. In this case, the environment dictates which option needs to be taken.

We are not aware of other work that discusses gadgets used by PSHAPE in case study 3, i.e., gadgets which initialize all registers used for passing parameters. We suspect this is likely due to the fact that not much research has been done on ROP automation and the other tools we know of are not made to handle gadgets this long. The exemplary gadget we show in Figure 5.5 consists of 29 instructions. Loading important registers starts with the fifth instruction, hence the useful part of the gadget is 25 instructions long[3]. The existence of such gadgets warrants further research regarding their availability. We leave this to future work.

[3]Remember that in case study 3 we were using only call-preceded gadgets, hence the whole gadget had to be used. If call-precedence is not required, the first four instructions can be skipped.

Chapter 6

Future Directions

This chapter discusses directions for future research. As the whole dissertation, its focus is the topic of ROP chain generation. With promising research on mitigations that greatly reduce the number of available gadgets, forcing exploit developers to use longer and more complex gadgets, automation will become essential. Future tools will need the ability to analyse even very long and complex gadgets which contain control-flow transfers and complex logical and arithmetical instructions.

We divide our ideas for improvement in short-term, which could be implemented in our current set of tools with relative ease, and long-term, where large portions of the code require rewriting. More specifically, we consider it likely that a different IR would have to be used for these extensions, due to the way VEX handles flags.

6.1 Short-term

Dynamic Parameter Generation

Due to restrictions of the environment it might not be possible to use certain characters, perhaps preventing the injection of parameters. In such a case, fully dynamic parameter generation at runtime is required. This can be achieved by combining arithmetic gadgets until the desired target value is reached. However, with potentially millions of gadgets, a strategy to filter out the most useful ones first and then combine the remaining ones in a smart way is required. A widely used alternative is to inject a parameter that has been transformed to remove illegal characters, but whose transformation can be easily reversed. For example, instead of injecting `1000h`, which contains null bytes, an exploit developer can inject `ffffffffffffefffh`, load this value into the target register, and then use a gadget that `not`s this register.

More Gadget Types

Many attacks we introduced in Chapter 3.2 rely on the existence of specific gadgets in the target binary. Adding routines to PSHAPE that help finding such gadgets would increase its usefulness even further. For example, JOP relies on a dispatcher gadget. Many attacks require a stack pivot gadget. Finding stack pivot candidates is easy using our summaries: any gadget whose postconditions ensure that a) `rsp` is changed by a user-specified number of bytes or b) `rsp` is set to another register. Similarly, support for whole attack techniques, such as COOP [138], could be added.

Side-Effect Calculation

When longer gadgets have to be used, the chance increases that a gadget, as a side effect, changes the value of a register used later on. Imagine a gadget that loads `11223344h` into a register and the next gadget has the side effect of adding `10h` to this register. The analyst will have to keep this in mind, hence, to ensure the register contains the correct target value, will have to inject `11223344h` − `10h`, i.e., `11223334h`. While this is a very simple example where the side effect is a basic mathematical operation, imagine side effects such as shifts, rotates, or logic instructions, such as `neg`. In a scenario like that it would be helpful if PSHAPE had the ability to suggest the value the exploit developer has to inject in order to end up with the correct target value after all transformations have been applied.

Load-All Gadgets

PSHAPE found interesting gadgets that load all registers used for passing parameters. While we have not found them being used in real exploits, it would be interesting to know how common such gadgets are. Using PSHAPE and its summaries it is simple to find them: any gadget with postconditions where all registers used for passing parameters are initialized with data relative to an exploit developer-controlled register are such candidates. This shows yet another advantage of our summaries, as they allow easy expansion of PSHAPE's features. Based on the summaries it was simple to extend PSHAPE to allow it to find the heuristic breakers we used in Chapter 5.2.

6.2 Long-term

Flag Handling

In Chapter 4.3.5 we showed that PSHAPE works very effectively in current, realistic environments, despite its restrictions. As we showed in Chapter 5, it can even be used to bypass real mitigation techniques. However, we are certain that CFI will become widely adopted

in the future. We are already seeing first steps taken with its integration in major compilers [36,101]. It will, however, suffer from early problems similar to ASLR, i.e., programs will use both CFI-aware and CFI-unaware libraries, allowing attackers to use arbitrary gadgets from CFI-unaware libraries. Over time, with increasing adoption, programs will be protected by CFI throughout, drastically limiting the attacker's choices of gadgets. In such an environment it will be vital for tools to be able to correctly analyse *all* gadgets. This requires them to be able to handle flags correctly. One way of achieving this is to treat a gadget with conditional instructions as several gadgets.

Consider the simple gadget `cmovne rax, rbx ; ret`. This gadget can be considered as two gadgets: one gadget copies `rbx` to `rax` and then returns, the other one just returns. Therefore, this gadget results in the two summaries shown in Figure 6.1.

```
PRE: ZF = 0
POST: rax = rbx
POST: rsp = rsp-8
```

```
PRE: ZF = 1
POST: rsp=rsp-8
```

Figure 6.1: Summaries of a `cmovne rax, rbx ; ret` gadget.

More Code-Reuse Techniques

Incorporating more code-reuse techniques such as JOP will be necessary in the future. Intel recently introduced a mitigation against ROP in hardware [85], which uses a shadow stack, effectively eliminating ROP, and limiting the legal targets of indirect jumps and calls, making JOP and other forms of code-reuse difficult. Taking JOP as a starting point, we plan to extend PSHAPE to be able to use techniques beyond ROP.

Chapter 7

Conclusion

With DEP preventing the execution of injected code, ROP has become a vital part of modern memory corruption exploits. Promising mitigations against ROP, such as a hardware shadow stack or SafeStack, are not merely confined to academia anymore, rather they are gradually being deployed and applied in real-world systems. This process, however, is slow, and until these mitigations find wide adoption, ROP remains a big threat. Researchers create attacks almost as quickly as mitigations, turning the topic of ROP into an arms race. At the same time, despite ROP's importance, tools supporting the process of developing ROP exploits are lacking in practical features, even though this is a task regularly attempted for non-malicious purposes like evaluating mitigation techniques, penetration testing, or whitehat hacking competitions. Many mitigations against ROP attacks drastically reduce the number of available gadgets and force exploit developers to use longer, more complex gadgets. However, even in current environments, where such mitigations are not present yet, existing tools do not work as expected and are mostly limited to gadget extraction. This makes ROP exploit development a predominantly manual task, which is both time-consuming and error-prone.

In this dissertation, we presented novel techniques to ROP exploit mitigation as well as support for ROP exploit development.

7.1 Summary and contributions

As noted in the thesis statement in Chapter 1.2, this dissertation combines various techniques to make automated ROP chain generation feasible.

We presented the following contributions:

- ROPocop, an environment restriction which detects abnormal program behaviour that can be linked to code-reuse attacks. It detects real-world ROP exploits, but we also

used it to show that heuristic-based mitigations can be bypassed in a fully automated manner by PSHAPE.

- PSHAPE, a framework geared towards practical support for ROP exploit development. It creates semantic summaries of gadgets and assembles them to a ROP chain which initializes all registers used for passing parameters to functions and ensures the gadgets are safe to use. It works in realistic, current scenarios but also under constraints imposed by mitigations such as ROPocop.

- GaLity, a set of metrics that measure gadget quality. It allows sorting gadgets by their usefulness and therefore results in a vast reduction of the search space.

ROPocop reliably detects ROP exploits and works best when the attacker is not aware of ROPocop's presence. Since it is a heuristic, however, an aware attacker can use long gadgets, or incorporate heuristic breakers into the ROP chain to bypass it. While this is not trivial to do manually, we show how such a process can be automated. To this end, we developed PSHAPE.

PSHAPE's ability to create functioning gadget chains fully automatically, even when mitigations are deployed, greatly improves on the current state of the art. Our evaluations and case studies showed its effectiveness and versatility. Apart from evaluating it against a regular, unprotected system, we also applied it to three use cases, where realistic mitigations put certain restrictions on the gadgets PSHAPE could use: i) We used PSHAPE to build ROP chains using only gadgets of length five or more to evade heuristics-based systems. ii) We used PSHAPE to find heuristic breakers, again, bypassing heuristic-based systems in a fully automated way. iii) We used PSHAPE to create ROP chains using only call-preceded gadgets, bypassing mitigations that enforce that every `ret` has to return to after a `call` instruction. In use case iii), PSHAPE also discovered gadgets that load all registers used for passing parameters. Such gadgets are interesting because they can reduce the number of gadgets required in an exploit to one, which warrants further research about their prevalence. Furthermore, PSHAPE offers semantic gadget summaries, a concise description of how a gadget affects memory and registers. These summaries are much easier and quicker to understand for analysts and simplify finding gadgets that achieve certain tasks. This makes them a great help to exploit developers in stages preceding ROP chain generation.

While PSHAPE, at this point, cannot use gadgets that contain instructions which change behaviour depending on flags, we have proposed a solution on how to handle these cases in future work. This change would allow PSHAPE to find and use even more gadgets, which will be important when mitigations that reduce the number of available gadgets are widely deployed. However, as our evaluation and case studies showed, for current and certain future scenarios this is rarely required and provides little benefit.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pages 340–353, Alexandria, VA, November 2005.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 51–66, 2009.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7, 1996.

[4] Starr Andersen and Vincent Abella. Changes to functionality in windows xp service pack 2 - part 3: Memory protection technologies, August 2004.

[5] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser JIT engines. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1342–1353, 2014.

[7] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, August 2014. USENIX Association.

[8] Arash Baratloo, Navjot Singh, and Timothy K. Tsai. Transparent run-time defense against stack-smashing attacks. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 251–262, 2000.

[9] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2014.*, 2015.

[10] Dennis Batchelder, Joe Blackbird, David Felstead, Paul Henry, Jeff Jones, Aneesh Kulkarni, John Lambert, Marc Lauricella, Ken Malcolmson, Matt Miller, Nam Ng, Daryl Pecelj, Tim Rains, Vidya Sekhar, Holly Stewart, Todd Thompson, David Weston, and Terry Zink. Microsoft security intelligence report volume 16, 2013.

[11] Eli Bendersky. pyelftools. `https://github.com/eliben/pyelftools`.

[12] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 158–168, 2006.

[13] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[14] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[16] David Bigelow, Thomas Hobson, Robert Rudd, William W. Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 268–279, 2015.

[17] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society.

[18] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[19] Erik Bosman and Herbert Bos. Framing signals - a return to portable shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 243–258, Washington, DC, USA, 2014. IEEE Computer Society.

[20] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[21] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[22] Mark Brand and Chris Evans. Significant flash exploit mitigations are live in v18.0.0.209. `https://googleprojectzero.blogspot.de/2015/07/significant-flash-exploit-mitigations_16.html`.

[23] Brandon Bray. Compiler security checks in depth. `http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx`, February 2002.

[24] Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, abs/1602.04056, 2016.

[25] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[26] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[27] Ero Carrera. pyelftools. `https://github.com/erocarrera/pefile`.

[28] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[29] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th international Conference on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[31] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[32] Xiaobo Chen. Aslr bypass apocalypse in recent zero-day exploits. `https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`.

[33] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[34] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*, pages 409–417, 2001.

[35] Clang Team. Clang 4.0 documentation: Safestack. http://clang.llvm.org/docs/SafeStack.html.

[36] Clang Team. Control flow integrity design documentation. http://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.

[37] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. ACM.

[38] corelanc0d3r. mona.py. https://github.com/corelan/mona.

[39] corelanc0d3r. Universal dep/aslr bypass with msvcr71.dll and mona.py. https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/, July 2011.

[40] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to protect return addresses from attack. *SIGARCH Computer Architecture News*, 33(1):65–72, 2005.

[41] Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, 1998.

[42] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 763–780, 2015.

[43] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 243–255, 2015.

[44] Stefano Cristalli, Mattia Pagnozzi, Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. Micro-virtualization memory tracing to detect and prevent spraying attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 431–446, Austin, TX, August 2016. USENIX Association.

[45] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[46] Roman Danyliw. Ms-sql server worm. http://www.cert.org/historical/advisories/CA-2003-04.cfm.

[47] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: hardware-assisted flow integrity extension. In

Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015, pages 74:1–74:6, 2015.

[48] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[49] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security*, SEC'14, pages 401–416, Berkeley, CA, USA, 2014. USENIX Association.

[50] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[51] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nurnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[52] Theo de Raadt. Exploit mitigation techniques. `http://www.openbsd.org/papers/ven05-deraadt/index.html`, 2005.

[53] Long Le Dinh. Ropeme - rop exploit made easy. `https://github.com/packz/ropeme`.

[54] Chad Dougherty, Jeffrey Havrilla, Shawn Hernan, and Marty Lindner. W32/blaster worm. `http://www.cert.org/historical/advisories/CA-2003-20.cfm`.

[55] Paul Ducklin. Anatomy of an exploit - inside the cve-2013-3893 internet explorer zero-day - part 2. `https://nakedsecurity.sophos.com/2013/10/25/anatomy-of-an-exploit-inside-the-cve-2013-3893-internet-explorer-zero-day-part-2/`, October 2013.

[56] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[57] Andrew Edwards, Hoi Vo, Amitabh Srivastava, and Amitabh Srivastava. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft, 2001.

[58] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 781–796, 2015.

[59] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin C. Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained

control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 901–913, 2015.

[60] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the elf ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, Washington, D.C., August 2015. USENIX Association.

[61] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[62] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets - towards a metric to measure gadget quality. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, pages 155–172, 2016.

[63] Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos K. Ispoglou, Mathias Payer, and Eric Bodden. PSHAPE: automatically combining gadgets for arbitrary method execution. In *Security and Trust Management - 12th International Workshop, STM 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings*, pages 212–228, 2016.

[64] Andreas Follner and Eric Bodden. ROPocop - Dynamic mitigation of code-reuse attacks. *Journal of Information Security and Applications*, 29:16 – 26, 2016.

[65] Michael Frantzen and Michael Shuey. Stackghost: Hardware facilitated stack protection. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, 2001.

[66] Martin Gallo. Agafi. `https://github.com/CoreSecurity/Agafi`.

[67] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 396–405, 2014.

[68] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[69] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, Bellevue, WA, 2012. USENIX.

[70] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[71] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 417–432, Berkeley, CA, USA, 2014. USENIX Association.

[72] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119, Austin, TX, August 2016. USENIX Association.

[73] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 47–58, 2016.

[74] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 65–72, New York, NY, USA, 2006. ACM.

[75] István Haller, Enes Göktas, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 341–350, 2015.

[76] István Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 517–528, 2016.

[77] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: where'd my gadgets go? In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 571–585, 2012.

[78] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2012. USENIX.

[79] Allen Householder. "code red" worm exploiting buffer overflow in IIS indexing service dll. `http://www.cert.org/historical/incident_notes/IN-2001-08.cfm`.

[80] Michael Howard, Matt Miller, John Lambert, and Matt Thomlinson. Windows ISV software security defenses. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`, December 2010.

[81] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 969–986, 2016.

[82] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[83] Immunity Inc. White phosphorus exploit pack sayonara aslr dep bypass technique. https://web.archive.org/web/20130317001810/http://www.whitephosphorus.org/sayonara.txt, June 2011.

[84] Intel. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b, and 3c, June 2015.

[85] Intel. Control-flow enforcement technology preview, June 2016.

[86] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[87] Mateusz Jurczyk. One font vulnerability to rule them all #2: Adobe reader rce exploitation. http://googleprojectzero.blogspot.de/2015/08/one-font-vulnerability-to-rule-them-all.html, August 2015.

[88] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual international Symposium on Computer Architecture*, ISCA '12, pages 94–105, Washington, DC, USA, 2012. IEEE Computer Society.

[89] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[90] Tim Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universität Bochum, 2009.

[91] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 147–163, 2014.

[92] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[93] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 81–96, 2015.

[94] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[95] Xiaoning Li and Peter Szor. Emerging stack pivoting exploits bypass common security. http://blogs.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security, May 2013.

[96] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to make ASLR win the clone wars: Runtime re-randomization. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[97] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 280–291, 2015.

[98] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[99] Giorgi Maisuradze, Michael Backes, and Christian Rossow. What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 139–156, Austin, TX, August 2016. USENIX Association.

[100] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6. `http://www.x86-64.org/documentation/abi.pdf`, October 2013.

[101] Microsoft. Control flow guard. `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`.

[102] Microsoft. Software defense: safe unlinking and reference count hardening. `https://blogs.technet.microsoft.com/srd/2013/11/06/software-defense-safe-unlinking-and-reference-count-hardening/`.

[103] Microsoft. Stack allocation. `https://msdn.microsoft.com/en-us/library/ew5tede7.aspx`.

[104] Microsoft. VirtualAlloc function. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366887%28v=vs.85%29.aspx`.

[105] Microsoft. x64 architecture. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff561499%28v=vs.85%29.aspx`.

[106] Microsoft. Enhanced mitigation experience toolkit 5.5 beta user guide, September 2015. [accessed: 2015-10-20].

[107] Jae-Won Min, Sung-Min Jung, Dong-Young Lee, and Tai-Myoung Chung. Jump oriented programming on windows platform (on the x86). In Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, Ana Maria A. C. Rocha, David Taniar, and Bernady O. Apduhan, editors, *ICCSA (3)*, volume 7335 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2012.

[108] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[109] Msfrop. https://www.offensive-security.com/metasploit-unleashed/msfrop/.

[110] Danny Nebenzahl, Mooly Sagiv, and Avishai Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. *IEEE Trans. Dependable Secur. Comput.*, 3(1):78–90, January 2006.

[111] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[112] Anh Quynh Nguyen. Capstone: Next generation disassembly framework. http://www.capstone-engine.org/BHUSA2014-capstone.pdf.

[113] Anh Quynh Nguyen. Optirop. https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-WP.pdf.

[114] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings*, pages 103–117, 2011.

[115] Offensive Security. Disarming and bypassing emet 5.1. https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/.

[116] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 121–138, Austin, TX, August 2016. USENIX Association.

[117] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[118] Pakt. Ropc. https://github.com/pakt/ropc.

[119] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[120] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX.

[121] S. H. Park, Y. J. Han, S. j. Hong, H. C. Kim, and T. M. Chung. The dynamic buffer overflow detection and prevent ion tool for yindows executables using binary rewr iting. In *The 9th International Conference on Advanced Communication Technology*, volume 3, pages 1776–1781, Feb 2007.

[122] PaX Team. Pax aslr. https://pax.grsecurity.net/docs/aslr.txt, 2003.

[123] PaX Team. Pax noexec. https://pax.grsecurity.net/docs/noexec.txt, 2003.

[124] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, pages 138–154, 2016.

[125] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA'15: 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 7 2015.

[126] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th International Conference on Virtual Execution Environments, VEE 2011, Newport Beach, CA, USA, March 9-11, 2011 (co-located with ASPLOS 2011)*, pages 157–168, 2011.

[127] Mathias Payer and Thomas R. Gross. String oriented programming: When aslr is not enough. In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, pages 2:1–2:9, New York, NY, USA, 2013. ACM.

[128] Peter Pi. Unpatched flash player flaw, more pocs found in hacking team leak. `http://blog.trendmicro.com/trendlabs-security-intelligence/unpatched-flash-player-flaws-more-pocs-found-in-hacking-team-leak/`, July 2015.

[129] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in COTS C++ binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[130] Manish Prasad and Tziker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[131] r41p41. Defeating emet 5.2 & 5.5. `http://casual-scrutiny.blogspot.de/2015/03/defeating-emet-52.html`.

[132] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. Intel memory protection extensions (intel mpx) enabling guide, April 2015.

[133] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 169–186, 2009.

[134] Dennis M. Ritchie. The development of the c language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 201–208, New York, NY, USA, 1993. ACM.

[135] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[136] Jonathan Salwan. Ropgadget. `https://github.com/JonathanSalwan/ROPgadget`.

[137] Sascha Schirra. Ropper - rop gadget finder and binary information tool. `https://scoding.de/ropper/`.

[138] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[139] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 88–108, 2014.

[140] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.

[141] Jeff Seibert, Hamed Okkhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 54–65, 2014.

[142] Fermin J Serna. Cve-2012-0769, the case of the perfect info leak, 2012.

[143] Fermin J. Serna. The info leak era of software exploitation, 2012.

[144] Hovav 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 '07, pages 552–561, New York, NY, USA, 2007. ACM.

[145] Robert W. Shirey. Internet security glossary, version 2. RFC 4949, RFC Editor, 2007.

[146] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[147] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[148] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[149] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[150] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[151] Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 954–968, 2016.

[152] Pablo Solé. Deplib. `https://www.immunitysec.com/downloads/DEPLIB.pdf`.

[153] Alexander Sotirov. Heap feng shui in javascript, 2007.

[154] Axel Souchet. rp++. `https://github.com/0vercl0k/rp`.

[155] Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, January 1989.

[156] Blaine Stancill, Kevin Z. Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, and Ahmad-Reza Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets. In *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, pages 62–81, 2013.

[157] Pseudo Sue. Roper. `https://github.com/oblivia-simplex/roper`.

[158] Ady Tal. Intel software development emulator. `https://software.intel.com/en-us/articles/intel-software-development-emulator`.

[159] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 256–267, 2015.

[160] Jack Tang. Isolated heap for internet explorer helps mitigate uaf exploits. `http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/`, July 2014.

[161] Jack Tang. Mitigating uaf exploits with delay free for internet explorer. `http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/`, July 2014.

[162] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[163] Arjan van de Ven. New security enhancements in red hat enterprise linux v.3, update 3. `http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf`, August 2004.

[164] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. ACM.

[165] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 934–953, 2016.

[166] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[167] Aurélien Wailly. nrop. `https://github.com/awailly/nrop`.

[168] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and communications security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[169] Mark Yason. Understanding ie's new exploit mitigations: The memory protector and the isolated heap. `https://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap/`, August 2014.

[170] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[171] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pages 145–156, 2010.

[172] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[173] Chao Zhang, Dawn Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. Vtrust: Regaining trust on virtual calls. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[174] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[175] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.

[176] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. HDROP: detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings*, pages 172–186, 2014.

Academic Résumé

July 2012 - December 2016
Doctoral studies at the chair of Prof. Dr. Eric Bodden, Secure Software Engineering Group, Department of Computer Science, Technische Universität Darmstadt

September 2010 - April 2012
Studies of information management and IT security at FH Technikum Wien. Completed with the Master of Science academic degree. Graduated with honours.

September 2008 - June 2010
Studies of computer science at FH Technikum Wien. Completed with the Bachelor of Science academic degree.