Dynamic Forensic Techniques for Rebuilding Code Reuse Attacks Payload

Submitted by

Mohamad Ridzuan

Thesis Advisors

Jianying Zhou
Flavio Toffalini

Information Systems Technology and Design

A thesis submitted to the Singapore University of Technology and Design in fulfillment of the requirement for the degree of Master of Science in Security by Design

2021

Declaration

I, Mohamad Ridzuan, declare that this thesis titled, "Dynamic Forensic Techniques for Rebuilding Code Reuse Attacks Payload" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

Information Systems Technology and Design

Master of Science in Security by Design

Dynamic Forensic Techniques for Rebuilding Code Reuse Attacks Payload

by Mohamad Ridzuan

Code Reuse Attacks using Return Oriented Programming (ROP) techniques are becoming more prevalent as it bypasses defences such as Data Execution Prevention (DEP). Existing defences against ROP attacks mainly focus on prevention and detection. They require side information or impose significant overhead, which limits their practicality.

In this thesis, we focus on post-incident scenarios where we attempt to rebuild the payload used for the ROP attack. In contrast to existing works, our solution can be immediately deployed to end-users since it does not rely on side information which is rarely provided in practice. We demonstrate and evaluate our solution using programs vulnerable to ROP attacks, which demonstrates the feasibility and performance of our solution.

Acknowledgements

This thesis was carried out during the period from November 2020 to August 2021. The challenges along the way led to an enriching experience. The thesis has provided me with experiences and knowledge that I could not have obtained elsewhere. It is truly a great feeling to complete this study.

I would like to thank my supervisor Professor Jianying Zhou for helping me with the problem statement, overall direction, and flow of my work. I would also like to thank Flavio Toffalini, my co-supervisor, for his significant inputs and for motivating me along the way.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

Digital platforms have become part of our everyday lives. Various platforms, ranging from servers, desktop computers, and mobile devices. They help us in daily tasks, such as working or communicating with our beloved. Unfortunately, the widespread computing platforms also increased the likelihood of founding software vulnerabilities, thus increasing the attack surface. Overall, the cause of vulnerabilities is twofold. First, the number of software bugs increases exponentially with the codebase. Second, programs are often written by developers who might not be well-versed in security. As a result, the software does operate as intended but lacks security aspects. In this scenario, an adversary can exploit a program to trigger an action never intended by the developer. In particular, software exploits have been increasing in either frequency and impact. Once a vulnerability is found, exploiting a program becomes straightforward as just inputting a too-long string of text.

Despite extensive research and many proposed solutions, vulnerabilities *e.g.*, buffer overflow [28], heap overflow [3] are existing in modern-day programs. These vulnerabilities have led to the development of the security feature, Data Execution Prevention (DEP) [11]. DEP prevents pages with write permissions in the memory from being executed, making it hard for an adversary to redirect control flow to the injected malicious code.

Code-Reuse attacks (CRA) are well-known techniques to bypass the DEP defence mechanism. Instead of code injection, the adversary bypasses DEP by using pieces of code that are already part of the program memory space to activate the malicious behaviour. Return-into-libc [33] is one CRA technique in which an adversary manipulate the return address in the stack to an address of the library function *e.g.*, the function `system` from the library *libc*.

From either research or industry, much effort is spent to mitigate such vulnerabilities [1, 17, 38, 4, 9, 10, 24], however, relatively few works try to analyse the attack from a forensic perspective [19]. In this thesis, we explore the new analysis techniques for code-reuse attacks (especially ROP-chains). Intuitively, having vital tools to analyse the payloads help an analyst to understand how an attack has been carried out, and finally aids the development of better defence systems. In Section 1.1, we expand the motivation of this thesis.

1.1 Motivation

In this section, we discuss the research motivation of this thesis. We first focus on the prevalence of Code Reuse Attacks (Section 1.1.1). We then discuss the status of the

current defence and technique to analyse Code Reuse Attacks (Section 1.1.2 and 1.1.3). Finally, we discuss the related works for our thesis (Section 1.2).

1.1.1 Increase prevalence of Code Reuse Attacks

The competition between attackers and defenders has led to CRA techniques such as Return-into-libc [33], Return-Oriented Programming (ROP) [6] and Jump-Oriented Programming [5]. These, in turn, have encouraged much academic research. The first attack was detected in 2010 targeting Adobe PDF [37] using the ROP technique. At this time, many platforms had not implemented DEP, allowing attackers to launch code injection attacks. From then on, many ROP attacks have appeared [18, 26, 40]. Proprietary programs are usually the target for ROP attacks.

1.1.2 Code Reuse Attacks Defence Techniques Limitations

There exists a large number of proposals that aim to detect and mitigate Code Reuse Attacks [1, 17]. However, many of them have been bypassed in a never-ending attack-defence arms-race [38, 4]. Moreover, more advanced implementations cannot be easily adopted because the requirements are restricted. Some of these solutions require side information [9, 10, 24] *e.g.,* source code, debugging information, which is usually not provided in practice. Therefore, we tackle the problem from another perspective. Our approach is to let the attack happen and study it.

1.1.3 Digital Forensic Analysis

Mariano Graziano et al. [19] mentioned that ROP attacks are often underestimated. The issue that analyst may often encounter are Code Reuse Attacks are hard to detect, and finding the entry point can be very challenging. This information can be crucial for an analyst to be able to trace and determine the techniques used by the adversary to launch the attack. However, many code-reuse payloads are one-shot, *i.e.,* they disappear in memory after they are activated. Our technique, instead, tries to overcome this limitation and provide a reconstructed payload even after an attack has been carried out.

1.2 Related Works

In the following sections, we review the related works in ROP defences. We discuss on Control Flow Integrity (Section 1.2.1) and Memory Forensic Analysis (Section 1.2.2).

1.2.1 Control Flow Integrity

The execution in Code Reuse Attacks disrupts the original control flow unintended by the developer. Control flow integrity [2] pre-computes the boundary of the program execution to prevent the malicious attack from redirecting the control flow. Deriving the control flow graph can be challenging, especially for complex programs.

Hong Hu et al. [21] developed a system, μCFI, that performs static data flow analysis to identify constraining data from the program source code. It then runs in parallel with the program execution to parse the recorded constraining data. μCFI asynchronous checks for the target of control transfer instructions after the program execution delays the detection of the attack.

The limitation with μCFI is that it requires side information *i.e.,* the source code, to be recompiled to perform the static data flow analysis. Having access to the source code is usually not the case for proprietary programs which can be a target for ROP attacks. Ren Ding et al. [15] designed a runtime environment, PITTYPAT, a background process that performs online analysis of the program path executed. Their approach is distinct as it monitors the program during runtime using Intel Processor Trace for efficient recording, however, incurring overhead.

Many defences for ROP attacks are based on monitoring the program execution at a low instruction level. The majority of the proposals use dynamic binary instrumentation techniques, using frameworks *e.g.,* PinTool [29], DynamoRio [16]. This approach monitors the `ret` instruction and raises an alert if it detects irregularity.

ROPdefender [13] is a tool that enforces return address protection to detect ROP attacks that do not require any side information *e.g.,* source code, debugging information, during runtime. ROPdefender uses Pintool framework [25] which provides Just-In-Time binary instrumentation. It dynamically instruments `call` and `ret` instructions to store in the shadow stack. It then compares the shadow stack with the system's stack on every function exit to determine an ROP attack. The solution can be applied to complex multi-threaded programs and deployed on various platforms such as Windows and Linux. However, the overhead can be significant due to the additional instrumentation limiting their practical applicability.

ROP-Hunt [34] is also a tool that detects ROP attacks by leveraging similar instrumentation techniques with ROPdefender. The tool is mainly to detect and report ROP attacks during the program runtime. Rop-Hunt validates if the address of the `ret` instruction and the target address are from the same routine. It then determines if the length of the instruction sequence exceeds 7. Here, they have assumed the gadget size threshold base on the studies they conduct. As a result, they treat it as a possible attack. Their approach has 0 false positives. However, it incurs overhead due to the instrumentation and is limited to only x86 architecture.

These proposals mainly focus on detecting and preventing ROP attacks. They face particular challenges such as overhead due to the additional instrumentation or validation limiting their practicality. In our work, we adopt a different approach to find evidence and trace of the attack if an attack happens.

1.2.2 Memory Forensic Analysis

Another popular approach to analyse malicious programs is using memory forensic techniques to load the program to the memory and analyse the memory dump [19]. Preventive defences have yet to be widely used due to limitations such as performance overheads and accuracy. The detection approach is more favorable by operators today as the impact on performance is negligible.

Adversaries are moving towards data-only malware where it does not introduce any new code. It inserts control data structure into the system in the form of a ROP

chain, making it challenging to detect and prevent such attacks. Current systems [36] try to minimise the overhead incurred to detect such an attack as it can only detect the point in time when the malware starts executing.

Previous works focus on persistent code reuse attacks, while one-shot attacks (which are more common) disappear after their activation [39], and thus they cannot retrieve them from the memory dump. Our approach, instead, can also rebuild one-shot payloads; thus, it is more generic.

Chapter 2

Scope of work

The purpose of our work is to reconstruct a code-reuse attack payload from a known binary and its runtime trace without the need for side information.

The summary of our main work contributions are as follows:

1. Study if the runtime trace extracted from a process is sufficient to rebuild a code-reuse attacks payload.
2. Formalize the problem and propose an automatic payload recovering algorithm.
3. Study what payload information can or cannot be retrieved.

2.1 Outline

We organise the chapters as follows:

Chapter 3: We provide a background on the types of control flow attacks and mitigation.

Chapter 4: We present our solution of reconstructing the payload of a ROP attack without side information such as the source code.

Chapter 5: We evaluate the performance of our solution to reconstruct the payload and discuss some use cases.

Chapter 6: We discuss some of our challenges, shortcomings of our work, and future works.

Chapter 7: We conclude this thesis.

Chapter 3

Background

In this chapter, we provide a background on control flow attacks (Section 3.1) and defences (Section 3.3). Then, we describe the different techniques of Code Reuse attacks, Return-into-libc (Section 3.4.1) and Return-Oriented Programming (Section 3.4.2). The attacks operate at low-level assembly instructions and customise to specific processor architecture. We work on both 32 and 64-bit programs in this thesis.

3.1 Control Flow Attacks

In control flow attacks, an adversary's main goal is to redirect the program execution flow. A classic example would be the buffer overflow [28] attack where the adversary can overwrite the target address of a control flow instruction *e.g.,* a return instruction to redirect the control flow and trigger a malicious action *e.g.,* code execution by the shellcode.

There are two primary control flow attacks, (i) code injection and (ii) code-reuse attacks. Code injection requires injecting new code that needs to be loaded to memory and executed during runtime. In contrast, CRA uses instructions already residing in the memory space to bypass DEP [11]. It combines small pieces of code in the program to induce malicious behaviour.

The high level control flow graphs (CFG) of code injection and CRA are illustrated in Figure 3.1 and 3.2 respectively. The nodes n are linked with directed edges, and the edges indicate the possible control flow. Figure 3.1 illustrates the control flow graph for code injection attack. The malicious code consists of two nodes, n_a and n_b. They are not part of the original CFG, and the adversary needs to exploit the intended flow to redirect the flow to execute the malicious code. Figure 3.2 illustrates the control flow graph for code-reuse attack. In contrast to code injection, the adversary can inject the code pointers n_4 and n_1 by exploiting the vulnerability in n_3. This can lead to the unintended flow: $n_3 \rightarrow n_4 \rightarrow n_1$.

3.2 Code Injection Attack

To launch a code injection attack, the adversary needs to load the malicious code to the program address space in memory to be executed at a later stage. The adversary can encapsulate the malicious code into the data input of the vulnerable program. The vulnerable program can be lacking in checks *e.g.,* validating the length of the input

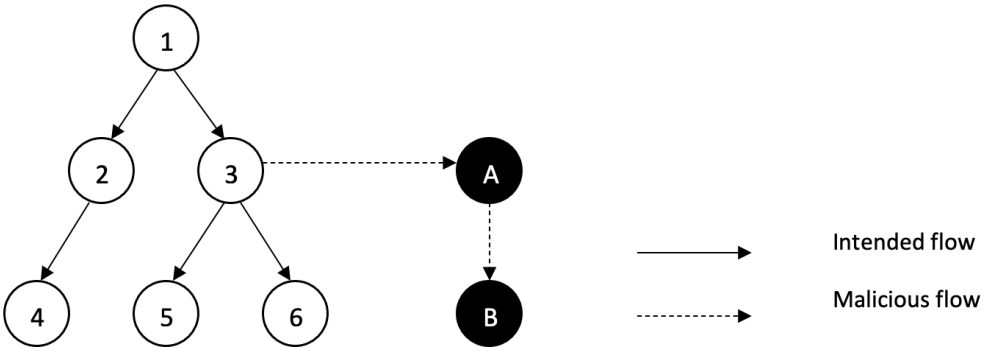

FIGURE 3.1: Control Flow Graph for Code Injection Attack.

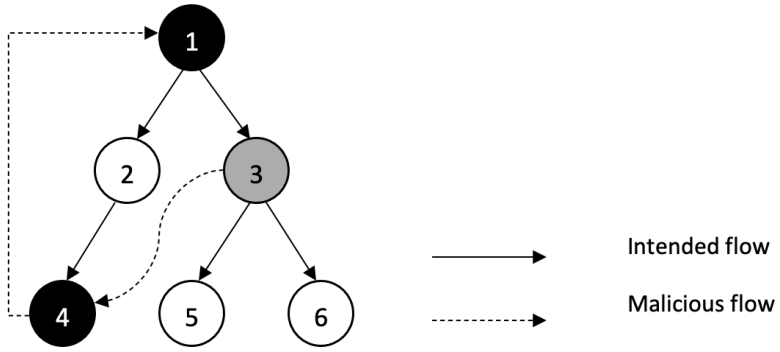

FIGURE 3.2: Control Flow Graph for Code Reuse Attacks.

string using `strcpy` function, allowing the adversary to cause a buffer overflow and inject the malicious code *i.e.*, shellcode into the program memory space.

Figure 3.3 illustrates the code injection attack. For this example, the buffer has a size of 100 bytes. The adversary runs the program and encapsulates the malicious code via the data input. As a result, the data input overwrites the fields above the buffer, including the base pointer and return address. Here the adversary crafts the data input to overwrite the base pointer with random values and the value of the return address to point to the location of the shellcode in the program memory. Consequently, the adversary can successfully use the vulnerable program to execute the malicious shellcode *e.g.*, opening a terminal for the adversary.

3.3 Data Execution Prevention

In the code injection attack we described in Section 3.2, the new code needs to be encapsulated to a variable, loaded, and written into the memory. Hence, data segments in the stack need to have read, write, and execute permissions, resulting in the vulnerability. Data Execution Prevention (DEP) is a defence mechanism in the operating system that prevents the program from executing code in the memory. An attempt to execute the

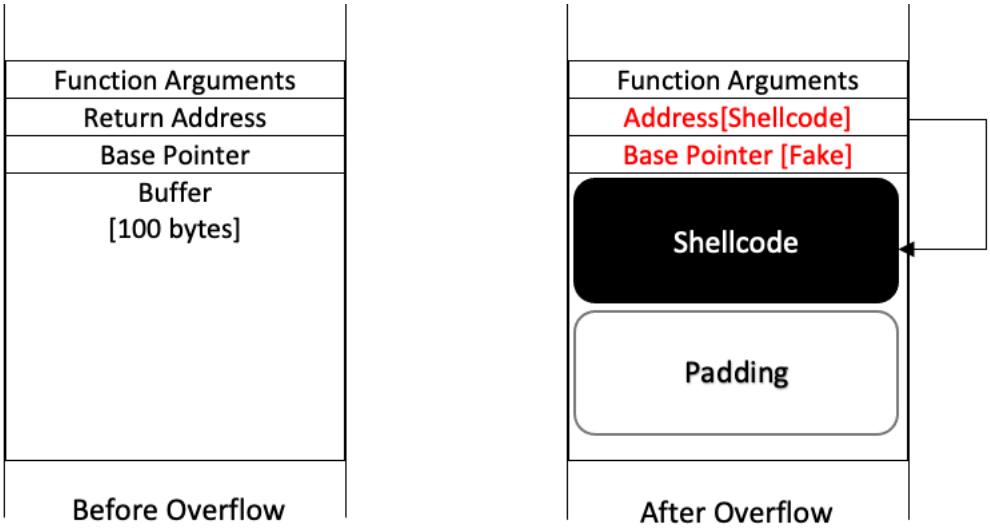

FIGURE 3.3: View of the stack for Code Injection attack.

code can result in the program throwing an exception and crashing. The general model is commonly known as Writable XOR Executable ($R \oplus W$). The main idea of $R \oplus W$ is to enforce the memory pages to either have write or execute permissions to prevent code injection attacks.

3.4 Code Reuse Attacks

In this section, we discuss the different techniques on code reuse attacks that overcome DEP defence, Return-into-libc (Section 3.4.1) and Return-Oriented Programming (Section 3.4.2), which is the focus of our thesis. After DEP [11] was implemented as a countermeasure against code injection, CRA techniques can be used to bypass the defence mechanism. Instead of injecting new malicious codes during program runtime, the adversary uses instructions already residing in program memory.

3.4.1 Return-into-libc

Return-into-libc [33] attack modifies the control flow by overwriting the target address of the `ret` instruction to point to a library function. Most commonly, it targets the `system()` function from the standard *libc* library which is linked to almost every process on the Linux platform. The `system()` function takes an input to be executed in a shell. For instance, `system("/bin/sh")` can open a terminal when executed where an adversary can carry out the malicious intent.

Figure 3.4 illustrates a classic return-into-libc attack. The adversary declares an environment variable `$SHELL` which contains the string `/bin/sh`. The adversary then provides an input that exceeds the buffer size causing an overflow resulting in the return addresses being overwritten with the addresses of the `system()`, `exit()` and the

`$SHELL` variables which are part of the payload. To construct the payload, the adversary requires the addresses that can be retrieved by debuggers or reverse-engineering tools *e.g.*, gdb, IDAPro.

However, there are limitations to the return-into-libc attacks. The attack is dependent on the *libc* functions; hence if it is omitted, the adversary cannot perform a reasonable attack. There also exists a challenge to carry out return-into-libc attacks on a 64-bit system where function arguments are passed using registers instead of the stack. The solution is to use `pop` instructions to load the arguments into the register to carry out the attack.

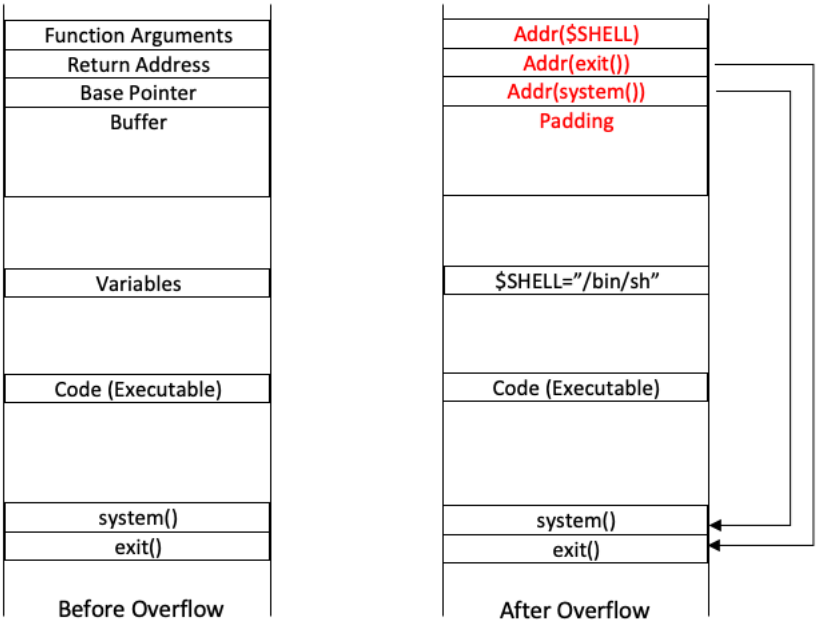

FIGURE 3.4: View of the stack for Return-into-libc attack.

3.4.2 Return-Oriented Programming

Return-Oriented Programming (ROP) [33] is an alternative CRA technique. It executes instruction sequences ending with a control transfer instruction *i.e.*, `ret`. Such a sequence of instructions is called a gadget and can be collected into chains. The adversary can redirect the control flow by transferring it from one gadget to another by manipulating the target address of the `ret` instruction.

Figure 3.5 illustrates how the adversary can redirect the program flow using ROP techniques. In step 1, the adversary uses a vulnerability *e.g.*, buffer overflow [28], to overwrite the return address of the function in order to gain control of the stack. In step 2, Return Address 1 redirects the program control flow to the first gadget. The first gadget transfers the control flow to the second gadget, and so on. This series of gadgets are collected into chains and are part of the payload.

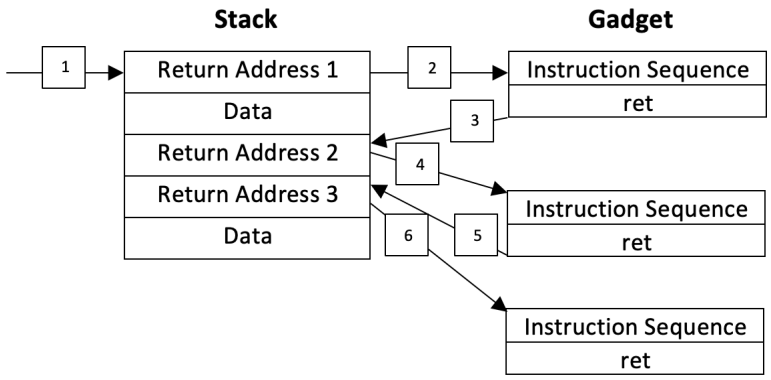

FIGURE 3.5: ROP attack workflow.

We consider the following indications to help us identify a gadget. The number of instruction sequences in a gadget is an essential factor for an ROP attack. Davi et al. [12] find that the number of instruction sequences in a gadget ranges from two to five instructions. DROP [8] indicates that the number of instruction sequences in a gadget is no more than five. The gadget can become very challenging to use as the number of instruction sequences increases [23]. It is also observed that it is unlikely to find ROP chains from the same function [34].

Chapter 4

Methodology

In this chapter, we describe the methodology adopted in this thesis. First, we define our threat model (Section 4.1). Then, we describe an overview of our approach (Section 4.2), retrieving gadgets (Section 4.3) and retrieving function address (Section 4.4). Finally, we describe our methods of binary instrumentation (Section 4.5) and our technique to rebuild the payload (Section 4.6).

4.1 Threat Model

In this work, we assume an unprivileged adversary that attempts to take control of a system through a code-reuse technique. The defender, instead, tries to rebuild the adversary actions and find evidence of the attack. We define adversary and defender requirements in sections 4.1.1 and 4.1.2, respectively.

4.1.1 Adversary Capabilities

The adversary is highly motivated and knowledgeable using ROP techniques. The end goal for the attacker is to redirect the program control flow to induce the malicious behaviour. The adversary is aware of memory vulnerabilities *e.g.,* buffer overflow to initiate the ROP attack.

We assume the target is vulnerable to information leakage allowing the adversary to infer the memory layout by inspecting the victim address space. Exploiting these vulnerabilities, the adversary can control the program stack to manipulate the existing return addresses to redirect the program flow. This assumption is consistent with published Code Reuse Attacks that use buffer overflow exploit to overwrite return addresses.

4.1.2 Defender Capabilities

We assume the vulnerable program is protected by DEP, defeating traditional code injection attacks on the defensive side. Here, we assume the defender is also the host computer. The defender can trace the process instructions at runtime. With this information, the defender can detect the malicious payload *i.e.,* gadgets, when the program is executed. In our work, the end goal of the defender is not to stop the attack but to rebuild the payload used to prove an attack happened. Finally, we do not require the source code of the victim program to reconstruct the payload. The binary program is sufficient for the defender to reconstruct the payload.

4.2 Overview

To better illustrate the roles of the actors, we discuss the following scenario. An adversary A aims to attack a defender D. A have access to $D's$ computer and discovered a vulnerable program. A then weaponizes the program with a deliverable payload. With the payload ready, A can deliver the attack to D in various manners *e.g.,* email, USB. Finally, A can execute the program coupled with the payload to achieve his goal.

After discovering the attack, D finds the program with the malicious payload. $D's$ main goal is to further investigate by tracing the instruction sequence and how the program flow is redirected due to the payload. In other words, $D's$ main goal is to find evidence the attack happened and reconstruct the payload.

There have been numerous proposals to detect and prevent ROP attacks [13, 35]. The adversary has to construct the payload to be coupled with the program to initiate the ROP attack. From a digital forensic analyst's point of view, it is vital to reconstruct the payload to assist with the investigation process. During the investigation, it is less likely that the analyst has access to the source code, making it challenging to determine the entry point and how the attack was carried out. To the best of our knowledge, no previous work was done to address this challenge.

Figure 4.1 shows the high-level workflow of the proposed solution. In our implementation, we need to (i) trace the instructions, (ii) detect the continuous gadgets, and (iii) reconstruct the payload. The workflow of the proposed solution involves three separate phases: an *offline program analysis*, *runtime analysis* and *post analysis*.

Offline program analysis – In this phase, we statically analyse the program. We retrieve the address of all the possible gadgets using ROPGadget (Figure 4.1 step 1) and the function addresses (Figure 4.1 step 2) to detect function calls that can be part of the attack. After retrieving the gadgets and function addresses, we save them in a database.

Runtime analysis – In this phase, we instrument the program using Pintool (Figure 4.1 step 3). It only traces the `ret` instruction by running the program with the payload and saves the trace in a log file.

Post analysis – In this phase, we cross validate the log file and the database and reconstruct the payload (Figure 4.1 step 4).

4.3 Retrieve Gadgets

In this section, we search and extract ROP gadgets from a binary program. Section 3.4.2 mentions ROP attack executes gadgets, and these gadgets can be collected into chains. We search, extract, and store these gadgets from the binary to reconstruct the payload (Section 4.6). The expected outputs are the starting address of the gadget and the series of instructions ending with `ret`. Gadgets used in the ROP attack can consist of unaligned instructions due to the variable length nature of the x86 architecture. Therefore, we can find gadgets across legitimate instructions. In the example of Figure 4.2, we have the instruction sequence of `mov; cmp; lea` at byte `C0`. However, when disassembling the next byte `D1`, a gadget is found ending with a `ret` instruction. Therefore,

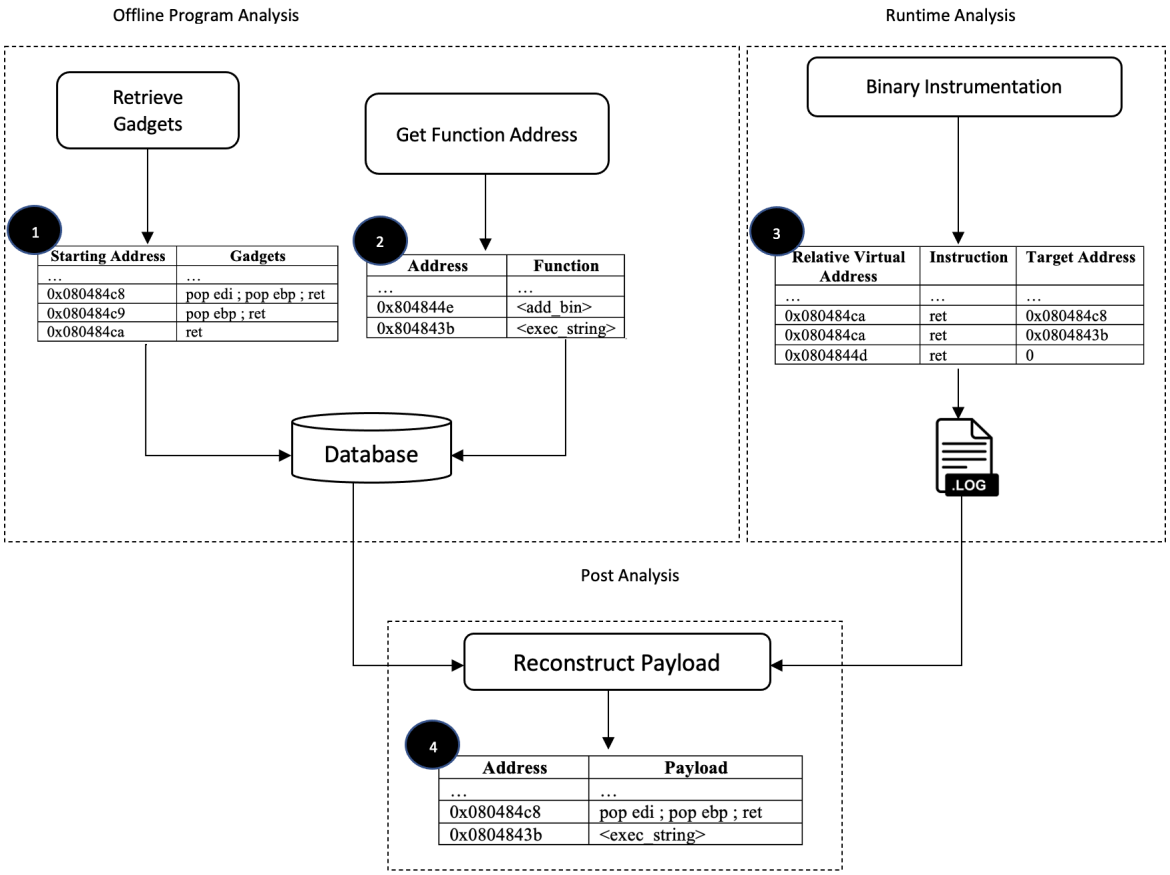

FIGURE 4.1: Workflow.

an adversary can build different gadgets by jumping into existing instructions; hence these gadgets must be discovered to reconstruct the payload.

We can consider the following algorithm to find gadgets from the binary:

1. We search the binary for `ret` instruction.
2. Once we find a `ret` instruction, we work backward to see if the previous instruction is valid. We reverse to the maximum amount of valid instructions.
3. We then record these instructions and repeat them till the end of the binary.

There are various tools available to automate the search for gadgets in the binary *e.g.,* ROPGadget [31], ROPium [27] and ROPper [32]. In our work, we incorporate ROPGadget [31] to search, extract and store gadgets in the binary into the database (Figure 4.1 step 1). ROPGadget is an open-source tool that the author and the community constantly maintain. It supports x86, x64, ARM, ARM64, PowerPC, SPARC, and MIPS architectures. The output of ROPGadget consists of the sequence of instructions ending with a `ret` or other instructions *i.e.,* `call` and `jmp` which redirects the control flow and the relative virtual address of the first instruction of the gadget, making it ideal.

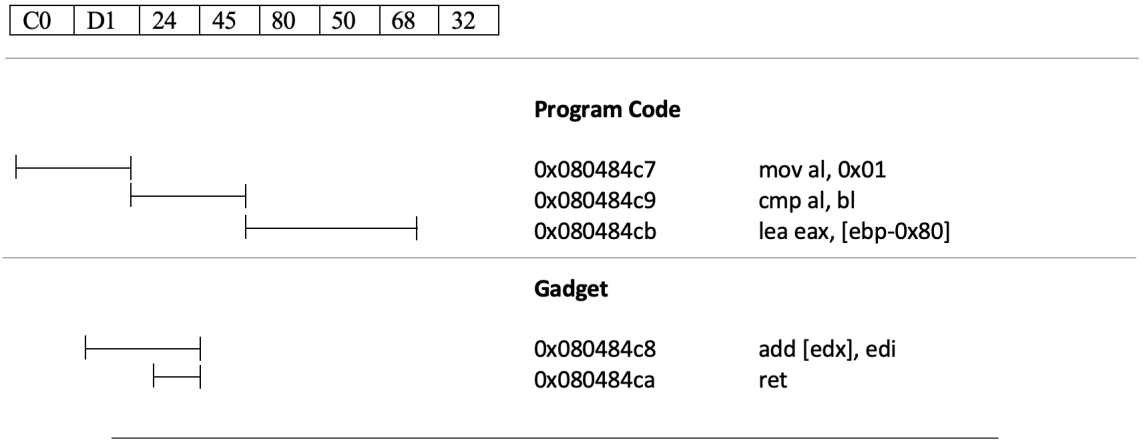

FIGURE 4.2: x86 Unaligned instruction sequence.

Listing 4.1 shows a snippet of disassembled instructions. The instruction sequence ended with a `ret` instruction (line 3) and the previous two instructions (line 1 and 2) are valid making them a potential gadget that can be used by the adversary. Listing 4.2 lists all the potential gadgets searched and extracted by ROPGadget from the binary with the starting address of the corresponding gadget.

```
1 0x080484c8: pop edi
2 0x080484c9: pop ebp
3 0x080484ca: ret
```

LISTING 4.1: Disassembled instructions.

```
1 0x080484c8 : pop edi, pop ebp, ret
2 0x080484c9 : pop ebp, ret
```

LISTING 4.2: ROPGadget Output.

4.4 Retrieving function addresses

Commonly, gadgets in ROP chains invoke functions as part of the attack. Therefore, in this section, we discuss techniques to collect function addresses to reconstruct the payload. In Section 4.3, we mentioned the use of ROPGadget to search and extract instruction sequence which ends in `ret`. It does not consider the function addresses. We consider two types of functions (i) functions internal to the binary and (ii) functions embedded in libraries either statically or dynamically loaded.

Internal Functions: Functions are usually not listed in a binary and there are many reverse engineering techniques to locate them [14, 20]. However, we assume that we can locate the function's address. For the sake of simplicity, in our work, we consider binaries that contain function symbols. This is reasonable since we do not deal with obfuscated binaries such as malware.

External Functions: Not all functions are stored in the binary. Functions *e.g., libc* are linked to the program to save space. When we see a `call` instruction to a function,

we cannot infer it directly. In this instance, it will invoke the Procedure Linkage Table (PLT). It will then jump to the address listed for the function in the Global Offset Table (GOT). PLT is a table in the binary file that stores the symbols that need a resolution *e.g.*, `printf()` (Listing 4.3 line 8), and this happens when a call to function is performed. GOT is a table that stores the address of the library functions that are dynamically bound to the calling program. Therefore, PLT and GOT are required to support dynamic library linking during runtime [22].

For the example in Listing 4.3, we first place a breakpoint at line 9 after the `printf()` function and run the program until we reach the breakpoint. In Listing 4.3 line 20, we can see that the instruction at `0x8048300` is a jump to the value stored at the memory address `0x804a000` that is the GOT entry in Listing 4.4 line 10. In Listing 4.3 line 24, we can see GOT is pointing to a new address `0xb7e6ced0`. In Listing 4.3 line $24-30$, we can see that we are inside the actual `printf()` function code after the address has been resolved by the dynamic linker.

```
1  Dump of assembler code for function main:
2     0x080483e4 <+0> :        push   ebp
3     0x080483e5 <+1> :        mov    ebp,esp
4     0x080483e7 <+3> :        and    esp,0xfffffff0
5     0x080483ea <+6> :        sub    esp,0x10
6     0x080483ed <+9> :        mov    eax,0x80484e0
7     0x080483f2 <+14>:        mov    DWORD PTR [esp],eax
8     0x080483f5 <+17>:        call   0x8048300 <printf@plt>
9     0x080483fa <+22>:        mov    eax,0x0
10    0x080483ff <+27>:        leave
11    0x08048400 <+28>:        ret
12 End of assembler dump.
13 (gdb) b *0x080483fa
14 Breakpoint 1 at 0x80483fa
15 (gdb) r
16 Starting program: ./hello
17
18 Breakpoint 1, 0x080483fa in main ()
19 (gdb) x/3i 0x8048300
20    0x8048300 <printf@plt>   :   jmp    DWORD PTR ds:0x804a000
21    0x8048306 <printf@plt+6> :   push   0x0
22    0x804830b <printf@plt+11>:   jmp    0x80482f0
23 (gdb) x/wx 0x804a000
24 0x804a000 <printf@got.plt>: 0xb7e6ced0
25 (gdb) x/5i 0xb7e6ced0
26    0xb7e6ced0 <printf>   :  push   ebx
27    0xb7e6ced1 <printf+1> :  sub    esp,0x18
28    0xb7e6ced4 <printf+4> :  call   0xb7f4af83
29    0xb7e6ced9 <printf+9> :  add    ebx,0x15811b
30    0xb7e6cedf <printf+15>:  lea    eax,[esp+0x24]
```

LISTING 4.3: Disassembled binary.

```
1 $ objdump -d -j .text ./hello | grep printf
2  80483f5: e8 06 ff ff ff        call   8048300 <printf@plt>
3 $ objdump -R hello
4
5 hello:     file format elf32-i386
6
7 DYNAMIC RELOCATION RECORDS
```

```
OFFSET    TYPE              VALUE
08049ff0  R_386_GLOB_DAT    __gmon_start__
0804a000  R_386_JUMP_SLOT   printf@GLIBC_2.0
0804a004  R_386_JUMP_SLOT   __gmon_start__
0804a008  R_386_JUMP_SLOT   __libc_start_main
```

LISTING 4.4: Global Offset Table Entries.

In our solution, we have incorporated objdump utility to disassemble the binary to search and extract the functions and addresses. Objdump utility provides a quick way to disassemble the binary and retrieve the information required for our solution. The result is stored in the database (Figure 4.1 step 2).

Ideally, we should be able to derive the following from a binary analysis:

1. Disassembled output of the program code *i.e.,* instructions, functions and modules.
2. Structure of the program *i.e.,* control and data flow
3. Data structures of the program *i.e.,* global and stack variables

However, this might not always be the case. In certain scenarios, the binary can be stripped where the symbol table is removed to save space or deter reverse engineering, primarily commercial software. In our work, this can pose a challenge if we are working with stripped binaries where it is not trivial to search and extract the functions. Y.David et al. [14] considered detecting procedure boundaries for stripped binaries to be an orthogonal problem due to the low amount of syntactic information.

4.5 Binary Instrumentation

In this section, we discuss how we trace the binary program at runtime. In Section 4.1.2, we had to assume that we had no access to the source code, and we make use of instrumentation techniques to trace the instructions during runtime. Instrumentation techniques are language independent and provide a machine-level view which complement the methods used in the *Offline Program Analysis* phase.

The approach for binary instrumentation is either statically (before runtime) or dynamically (during runtime). Dynamic instrumentation provides us with the following advantages:

1. There is no need to recompile or re-link. This is ideal in our work as we assumed we have no access to the source code.
2. Discovering code at runtime. As illustrated in Figure 3.5, the program is coupled with the payload and executed using the ROP technique to redirect the control flow. Dynamic instrumentation is ideal for tracing the program flow by attaching it to the running process.

We instrument all return instructions and the target address dynamically. This information is then written to the log file (Figure 4.1 step 3). There are various tools

available for binary instrumentation *e.g.,* PinTool [29], DynamoRio [16]. We have incorporated Pin Framework in our solution. Pin Framework provides a tool named Pintool [25] for the instrumentation of programs and traces all instructions that are executed sequentially. Pintool provides the following advantages making it ideal for our work:

1. Provides dynamic instrumentation. There is no need for the source code or recompilation which fits our requirements.

2. Programmable instrumentation. Pintool provides API in C++, allowing us to write customised instrumentation tools.

3. Multi-platform support. It supports x86 and x86-64 architecture. It also supports Mac OS, Linux, and Windows operating systems.

Pintool can operate in two modes, probe and just-in-time (JIT). JIT mode intercepts the instruction before it is executed, whereas probe mode traces the instructions that have been executed. We have implemented Pintool in probe mode in our solution as it traces the instructions with minimum overhead. The approach considers (i) `ret` instructions, (ii) instructions in the main ELF address space, and (iii) to output the relative virtual address. This approach is defined formally in Listing 4.5.

To only filter for `ret` instructions, we have called the following instrumentation functions `INS_IsRet(INS ins)` and `INS_IsSysret(INS ins)` (Listing 4.5 line 11) to determine if the instruction executed is a return instruction.

Dynamic libraries are usually loaded when the executable runs. To further reduce the overhead of tracing the entire ELF executable, we only trace the instructions in the main ELF address space. We call the following instrumentation function `IMG_IsMainExecutable(IMG img)` (Listing 4.5 line 23) to only trace instructions in the main ELF address space.

Application accesses the virtual memory i.e. Virtual Address (VA) and not physical memory during runtime. We need to retrieve the Relative Virtual Address (RVA) with respect to the ImageBase. ImageBase (Listing 4.5 line 26) here refers to the base address when the executable file is first loaded into memory. We can calculate the RVA with the following formula: $RVA = VA - ImageBase$ (Listing 4.5 line 27).

```
1  #include "pin.H"
2
3  void Instruction(INS ins, void* v) {
4      auto address = INS_Address(ins);
5
6      // Check that instructions address is inside main ELF image
7      if (address >= image_base_address &&
8          address < image_base_address + image_size) {
9
10         // Check if it is a ret instruction
11         if (INS_IsRet(ins) || INS_IsSysret(ins)) {
12
13             // Calculate the RVA
14             auto rva = address - image_base_address;
15             // Trace the ret instructions
16             INS_InsertCall(ins, rva);
17         }
```

```
18      }
19  }
20
21  void Image(img) {
22
23      if (IMG_IsMainExecutable(img)) {
24
25          // Store image base address and image size of the main ELF image
26          image_base_address = IMG_LoadOffset(img);
27          image_size = IMG_HighAddress(img)-IMG_LoadOffset(img);
28      }
29  }
30
31  int main() {
32      // Register function to be called to instrument instructions
33      INS_AddInstrumentFunction(Instruction);
34
35      // Instrument image loading
36      IMG_AddInstrumentFunction(Image);
37
38      // Start the program
39      PIN_StartProgram();
40  }
```

LISTING 4.5: Pintool instrumentation.

Listing 4.6 shows an expected output of the dynamic instrumentation by Pintool. The printout shows the RVA address of the `ret` instruction and the target address. For instance, the first line indicates that the process performed a jump (an edge in the execution flow) from RVA `0x80484ca` to `0x80484c8` through a `ret` instruction. However, in the case of ROP chain, the target points to either a *gadget* or a new *function address*, thus breaking the CFG as illustrated in Figure 3.2. In Section 4.6, we detail this idea in a well defined algorithm.

```
1  80484ca ret 80484c8
2  80484ca ret 804843b
3  804844d ret 0
```

LISTING 4.6: Pintool Trace.

4.6 Reconstructing the payload

In this section, we discuss the reconstruction of the payload (Figure 4.1 step 4) with the information retrieved from the *Offline Program Analysis* (Figure 4.1 step 1 and 2) and *Runtime Analysis* (Figure 4.1 step 3) phases.

The workflow to reconstruct the payload is define formally in Algorithm 1. In the example of Figure 4.3, we can see the different program flow between a normal and ROP execution flow. As discussed in Section 3.4.2, the adversary attempts to redirect the control flow using `ret` instructions. At runtime, the `ret` instructions of ROP code execution can be easily distinguished from the legitimate return instructions of a benign program by checking their targets. In the example, the `ret` instruction at `0x80484e2` is redirected to `0x804844e` which is an address of a function found in the database. At this stage, this can be a legitimate program flow or a candidate gadget. In the next `ret`

instruction at `0x8048489`, the program is again redirected to `0x80484c9` which is a gadget. As the control flow is transferred from one gadget to another, we can assume these instructions are part of the ROP chain and to be considered in reconstructing the payload.

Algorithm 1 takes in the trace logs from Pintool and the database containing the gadget and function addresses to produce the reconstructed payload. In line 2, we iterate through the program trace to retrieve the target address of the `ret` instruction. In line 3, we check if the target address matches an entry in the database. From line 4 to line 13, we check if the instruction is part of a legitimate program flow or a candidate gadget. If it finds the first gadget or function call (line 4), it takes the instruction to be part of the payload. If it finds the next control transfer instruction (`ret`) target address matches an entry in the database (line 7), it appends the instruction to be part of the payload. If it is neither of these conditions (line 10), we take it as a legitimate instruction. If it is a legitimate instruction, we clear the payload and continue to iterate the program trace.

The arguments for function calls must be taken into consideration when reconstructing the payload. We need to consider the `pop` instructions in the payload as they can contain the padding values that are pop-ed in the registers. We cannot infer the register's value directly, but the pop instructions can give information about the original payload size. For instance, given the function in Listing 4.7, we need to consider `arg1` and `arg2` passed to the function. In this instance, we would want the return address to be a `pop; pop; ret` gadget as shown in Figure 4.4 as there are two arguments. This would pop `arg1` and `arg2` from the stack and return to the next gadget. Finally, we check if the instruction is a `pop` instruction (line 14) to retrieve the token to append to the payload.

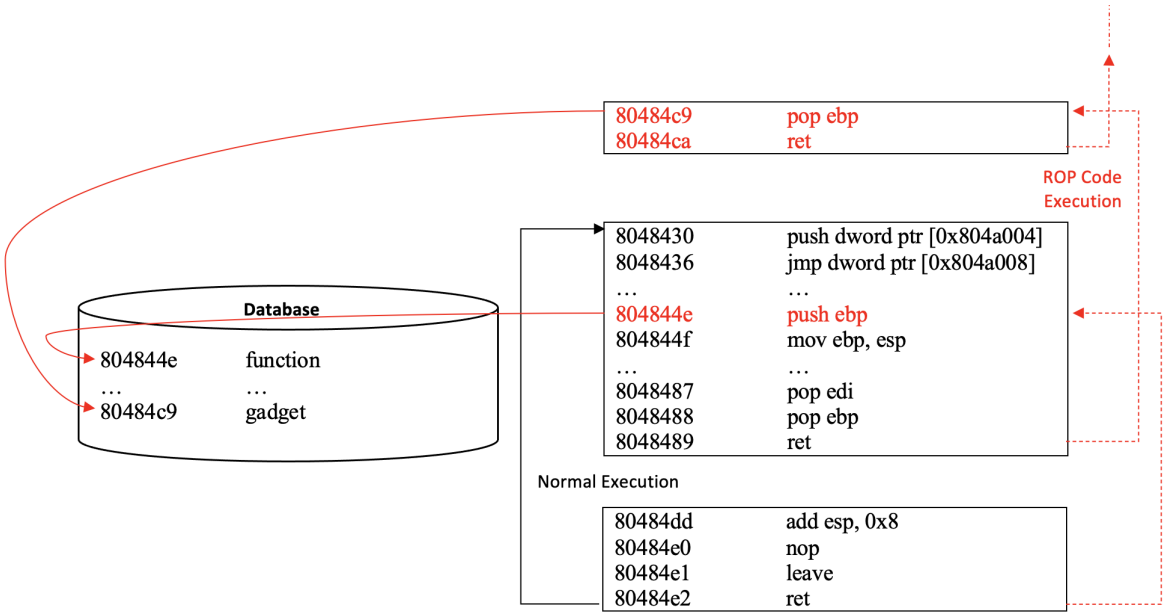

FIGURE 4.3: Validating ROP payload with database.

Algorithm 1: Reconstructing the Payload Algorithm.

```
function Construct Payload (Pin_Trace, Database);
Input  : Pin_Trace → Trace of Binary Instrumentation log
         Database → Gadgets and Functions
Output: Payload
for Address, Instruction ← Pin_Trace do
    if Instruction.Return_Address in Database then
        if Payload is Empty then
            Payload ← Instruction;
        end
        else if Previous_Instruction.Return_Address in Database then
            Payload ← Gadget;
        end
        else
            Payload.clear();
            continue;
        end
        if Instruction is "pop" then
            GetToken;
        end
    end
end
```

```
1 void function1(arg1, arg2) {
2     if(arg1 && arg2){
3         //do something
4     }
5 }
```

LISTING 4.7: Vulnerable Function.

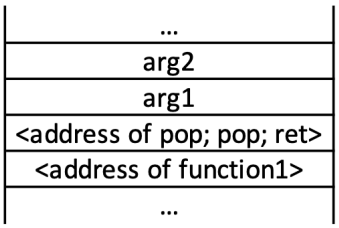

FIGURE 4.4: Gadget in the stack

Figure 4.1 illustrates how the payload can be reconstructed. In this scenario, from the program trace (Figure 4.1 step 3), traced dynamically from the binary. The record is then cross validated with the database and the return target address `0x080484c8` is pointing to a gadget `0x080484c8 pop edi; pop ebp; ret` (Figure 4.1 step 1).

The next instruction traced `0x080484ca ret 0x0804843b`, the return target address `0x0804843b` is pointing to the function address `0x0804843b exec_string`.

`pop` instructions *e.g.*, `pop edi`, `pop ebp` are instructions to load registers which are the function arguments and these arguments can be part of reconstructing the payload. From the disassembled instructions of the Pintool trace, we can then determine the size of the arguments by the registers as shown in Table 4.1.

TABLE 4.1: Register sizes.

8-byte registers	**4-byte registers**	**2-byte registers**	**1-byte registers**
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%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

Chapter 5

Results

In this chapter, we evaluate the performance of our solution in reconstructing the payload. We split our evaluation into two parts: (i) measure the performance of our solution to reconstruct the payload (Section 5.1) and (ii) technical discussion for our use cases (Section 5.2).

Our testing environment is Kali Linux 2020.4, Pintool version 3.17 and ROPGadget version 6.5. The programs we are exploiting are both 32 and 64 bit binaries. We verify using 6 binary programs vulnerable to ROP attacks for performance evaluation and use cases. These binary programs are from CTF challenges available on the web. The goal of our attacks is to retrieve the flags by running the exploits using ROP techniques on the binary.

We use Pwntools [30] to exploit the binaries using ROP techniques. Pwntools comes with exploit development libraries and uses Python, making the exploit writing as simple as possible. Using Python makes it easier to orchestrate running the binary program with the payload with other utilities *i.e.,* Pintool, objdump, and ROPGadget.

5.1 Performance Evaluation

In this section, we measure the precision and recall of our solution to reconstruct the payload. We construct the payload to be coupled with the vulnerable program using ROP techniques. We then run our solution and output the reconstructed payload and compare it to the actual payload to measure the performance of our solution.

The fields we consider to measure the performance of our solution are: (i) the number of function calls, (ii) the number of tokens that refer to the function arguments, and (iii) the number of gadgets that make up the ROP chain. We also consider the order of the reconstructed payload.

Table 5.1 shows the statistics for the payload for the 6 corresponding binaries and it's architecture as shown in the first two columns. After successfully constructing the payload to retrieve the flags from the binaries, we tabulated the numbers for the corresponding fields: (i) number of function calls, (ii) number of tokens, and (iii) number of gadgets in the last three columns. These values can be used to measure the performance of our solution.

Table 5.2 shows the statistics for the reconstructed payload. The results are obtained after running our solution. From the reconstructed payload, we tabulated the numbers of the three fields in the last three columns. We observe that the reconstructed payload is correct for the first three binaries, and the order is also a match. For the remaining binaries, we observe some discrepancies which we discuss in the use cases.

Table 5.3 shows the performance of our solution. We tabulated the number of false positives and false negatives in columns 2 and 3. We then calculate the performance using precision and recall in columns 4 and 5 respectively. Finally, we also consider the order of the reconstructed payload to measure the performance of our solution, as shown in the last column.

False-positive finding refers to gadgets in the reconstructed payload which is not part of the original payload. The gadget is included as part of the payload as it found a match in the log and database during the validation. There exist gadgets which do not end with a `ret` instruction *e.g.,* `jmp`. As a result, it is expected to have false-positive results. False-negative finding refers to the missing fields in the reconstructed payload. We provide more detail in the use case discussion.

TABLE 5.1: Statistics for the payload.

Program	*Arch.* *(32 or 64 bit)*	*Payload* *# of Functions*	*# of Tokens*	*# of Gadgets*
rop	32	3	3	2
babyrop	64	1	1	1
split	64	1	1	1
callme	64	3	9	3
write4	64	1	1	3
guess	64	1	6	7

TABLE 5.2: Statistics for the reconstructed payload.

Program	*Reconstructed payload* *# of Functions*	*# of Tokens*	*# of Gadgets*
rop	3	3	2
babyrop	1	1	1
split	1	1	1
callme	3	4	2
write4	1	1	2
guess	1	6	7

5.2 Case study 1

In this section, we discuss the *rop* binary. We use a simple and vulnerable 32 bit program. Listing 5.1 shows the exploit for the program, which will result in a shell for the adversary. First, the address of the gadgets is defined (Line 6 - 7). The function addresses are then defined (Line 8 - 10). The payload is then constructed (Line 17 - 27). The function arguments *i.e.,* tokens are also appended to the payload. The exploit initially starts the ROP execution using the buffer overflow (Line 13 - 14) to overwrite the return address.

TABLE 5.3: Precision and Recall for the reconstructed payload.

Program	*Results*				
	# of False +ve	*# of False -ve*	*Precision (%)*	*Recall (%)*	*Order*
rop	0	0	100	100	✓
babyrop	0	0	100	100	✓
split	0	0	100	100	✓
callme	2	8	77.78	46.67	✓
write4	0	1	100	80	✓
guess	0	0	100	100	✓

```
1  #!/usr/bin/python
2
3  import os
4  import struct
5
6  pop_ret = 0x80484c9
7  pop_pop_ret = 0x80484c8
8  exec_string = 0x0804843b
9  add_bin = 0x0804844e
10 add_sh = 0x0804848a
11
12 # First, the buffer overflow.
13 payload =  "A"*0x64
14 payload += "BBBB"
15
16 # The add_bin(0xdeadbeef) gadget.
17 payload += struct.pack("I", add_bin)
18 payload += struct.pack("I", pop_ret)
19 payload += struct.pack("I", 0xdeadbeef)
20
21 # The add_sh(0xcafebabe, 0x0badf00d) gadget.
22 payload += struct.pack("I", add_sh)
23 payload += struct.pack("I", pop_pop_ret)
24 payload += struct.pack("I", 0xcafebabe)
25 payload += struct.pack("I", 0xbadf00d)
26
27 payload += struct.pack("I", exec_string)
28
29 # Execute payload
```

LISTING 5.1: Program Exploit for case study 1.

Listing 5.2 shows the reconstructed payload after cross-referencing the database and the log. Line 1, 4 and 8 in Listing 5.2 are the function calls which appear in the log. Line 2 and 5 in Listing 5.2 are the gadgets which appear in the Pintool trace. Line 3, 6 and 7 in Listing 5.2 are the arguments for the function calls based on the register size as shown in Table 4.1. These instructions are considered to be part of the payload as the control flow is transferred from one gadget to another continuously.

```
1 0x804844e -> <add_bin>
2 0x80484c9 -> ['pop ebp', 'ret']
3 <token 4>
4 0x804848a -> <add_sh>:
```

```
0x80484c8 -> ['pop edi', 'pop ebp', 'ret']
<token 4>
<token 4>
0x804843b -> <exec_string>
```

LISTING 5.2: Reconstructed payload of case study 1.

5.3 Case study 2

In this section, we discuss the *guess* binary which is a 64-bit program. Listing 5.3 shows the construction of the payload to be coupled with the binary to spawn a shell for the adversary. The payload initially starts with the buffer overflow at line 2. Line 3 to 5 are the arguments for the function call. Line 6 to 10 are the addresses of the gadgets to be used in the payload. The address of the `syscall` is define in line 11. The payload is then constructed in line 13.

```
1  def generatePayload():
2      offset = b'a' * 120
3      data_address = p64(0x00000000006bc3a0) # data address to store the /bin/
       sh
4      bin_syscall = p64(generateString("/bin/sh"))
5      xor_rax_rax = p64(0x445950)
6      pop_rsi = p64(0x410ca3) # pop rsi ; ret
7      pop_rax = p64(0x4163f4) # pop rax ; ret
8      mov_rsi_rax = p64(0x47ff91) # mov qword ptr [rsi], rax ; ret
9      pop_rdi = p64(0x400696) # pop rdi ; ret
10     pop_rdx = p64(0x44a6b5)
11     syscall = p64(0x40137c)
12
13     payload = offset + pop_rax + bin_syscall + pop_rsi + data_address +
       mov_rsi_rax + pop_rax + p64(0x3b) + pop_rdi + data_address + pop_rsi +
       p64(0x0) + pop_rdx + p64(0x0) + syscall
14
15     return payload
```

LISTING 5.3: Program Exploit for case study 2.

Listing 5.4 shows the reconstructed payload after cross-referencing the database and the log. Line 1, 3, 5, 6, 8, 10 and 12 are the gadgets which appear in the Pintool trace. Line 2, 4, 7, 9, 11 and 13 are the arguments for the function calls based on the register size as shown in Table 4.1. Line 14 refers to the `syscall` which will spawn the shell for the adversary. These instructions are considered part of the payload as the control flow is continuously transferred from one gadget to another.

```
0x4163f4 -> ['pop rax', 'ret']
<token_8>
0x410ca3 -> ['pop rsi', 'ret']
<token_8>
0x47ff91 -> ['mov qword ptr [rsi], rax', 'ret']
0x4163f4 -> ['pop rax', 'ret']
<token_8>
0x400696 -> ['pop rdi', 'ret']
<token_8>
0x410ca3 -> ['pop rsi', 'ret']
<token_8>
```

```
0x44a6b5 -> ['pop rdx', 'ret']
<token_8>
0x40137c -> ['syscall']
```

LISTING 5.4: Reconstructed payload of case study 2.

Listing 5.5 shows the false positives for this case study. As discussed in Section 4.6, these are considered to be legitimate instruction as they are (i) `jmp` instruction or (ii) the previous instruction target address is not in the database. Therefore, these gadgets are not considered part of the payload as they can be categorised as legitimate instruction in the program flow, increasing the performance of our solution.

```
0x423772 -> ['mov r8, rax', 'jmp 0x4234aa']
<token_8>
<token_8>
<token_8>
0x44efde -> ['pop rbx', 'pop rbp', 'pop r12', 'jmp 0x45bad0']
<token_8>
0x400b4d -> ['pop rbp', 'jmp 0x400ab0']
0x401230 -> ['lea rdi, qword ptr [rsp + 0x60]', 'call 0x40dc40']
0x4112c2 -> ['cmp eax, -1', 'jne 0x411218', 'jmp 0x411291']
0x40ef39 -> ['add rsp, 8', 'ret']
0x4112c2 -> ['cmp eax, -1', 'jne 0x411218', 'jmp 0x411291']
0x413ae4 -> ['mov eax, dword ptr [rbx]', 'jmp 0x4138de']
0x4112c2 -> ['cmp eax, -1', 'jne 0x411218', 'jmp 0x411291']
0x400c26 -> ['mov dword ptr [rbp - 4], 1', 'jmp 0x400c3b']
0x460cbe -> ['mov r9d, eax', 'jmp 0x460bda']
0x462a9a -> ['jmp 0x46126a']
0x417a58 -> ['mov qword ptr [r12 + 0x28], rax', 'jmp 0x4179e1']
0x460cbe -> ['mov r9d, eax', 'jmp 0x460bda']
0x400c89 -> ['nop', 'leave', 'ret']
```

LISTING 5.5: False Positives for Case Study 2.

Chapter 6

Discussion

In this section, we provide a detailed discussion of our proposed solution.

6.1 Ret-less gadgets

Checkoway et al. [7] demonstrated that ROP attacks do not necessarily end with a `ret` instruction but with other type of control transfer instruction *i.e.,* `jmp`. This allows the use of a general-purpose register in place of the stack pointer to control the execution of gadgets and bypass stack integrity protections. For example, the gadget `pop eax; jmp edx` if `edx` is preset to the next gadget address. We can use Pintool to additionally instrument `jmp` instructions to trace the indirect branches. ROPGadget can search and retrieve gadgets ending with `jmp` instruction. We leave it to future work to determine if this approach is possible to reconstruct the payload.

6.2 False Positives

We encountered false-positive gadgets in our case studies as our solution relies on the gadgets searched and extracted by ROPGadgets. This is possible as these false positives traces are legitimate instructions. One can analyse the reconstructed payload and validate if it is a legitimate instruction or part of the ROP attack for false positives.

6.3 False Negatives

In addition, we did encounter false-negative gadgets in our case studies. These are gadgets that are not part of the reconstructed payload. We notice that these gadgets are usually part of the ROP chains that are contained in loops. To be able to detect them, we need to know the determined condition of the loop termination. We can do post-processing on the Pintool trace to analyse and simplify the output code. We again leave it to future work to determine if this approach is possible.

6.4 Solution Evaluation

Our solution can be deployed on both 32 and 64-bit architecture systems. It can be supported by Windows, Mac OS, and Linux. Pintool is compatible with all three operating

systems. ROPGadgets and our solution can be run using Python. Our thesis also focuses on post-incident where we let the attack happen and attempt to reconstruct the payload. In contrast with other works *e.g.,* ROPDefender where they focus more on detection and prevention. We again leave it to future work to complement with tools with such capabilities.

Chapter 7

Conclusion

There have been an increasing number of exploits in a program using ROP techniques. There are many existing proposals that focus on the detection and prevention of ROP attacks. These solutions work but have their limitations in practicability where there incur performance overhead or require side information.

With that in mind, in this thesis, we have proposed our solution, an approach that aims to reconstruct the payload. To prove the validity and practicability, we do not rely on side information such as the source code or debugging information as it leverages on dynamic instrumentation technique. We based our approach on runtime traces and try to rebuild the payload by checking the traces with the original binary. We tested our approach against 6 use cases, with a precision that ranges from 77% to 100% and a recall between 46% and 100%. We also plan to extend our approach for more generic code-reuse attacks and test it against real-world exploits.

Bibliography

[1] *A Technical Look at Intel's Control-flow Enforcement Technology*. URL: https://software.intel.com/content/www/us/en/develop/articles/technical-look-control-flow-enforcement-technology.html.

[2] Martin Abadi et al. "Control-Flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165. URL: https://doi.org/10.1145/1102120.1102165.

[3] Anonymous. "Once upon a free()". In: *Phrack Magazine* 57(9) (2001).

[4] Andrea Biondo, M. Conti, and D. Lain. "Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets". In: *NDSS*. 2018.

[5] Tyler Bletsch et al. "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. Hong Kong, China: Association for Computing Machinery, 2011, pp. 30–40. ISBN: 9781450305648. DOI: 10.1145/1966913.1966919. URL: https://doi.org/10.1145/1966913.1966919.

[6] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 385–399. ISBN: 9781931971157.

[7] Stephen Checkoway et al. "Return-Oriented Programming without Returns". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, pp. 559–572. ISBN: 9781450302456. DOI: 10.1145/1866307.1866370. URL: https://doi.org/10.1145/1866307.1866370.

[8] Ping Chen et al. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *Information Systems Security*. Ed. by Atul Prakash and Indranil Sen Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 163–177. ISBN: 978-3-642-10772-6.

[9] Tzi-Cker Chiueh and Fu-Hau Hsu. "RAD: a compile-time solution to buffer overflow attacks". In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 409–417. DOI: 10.1109/ICDSC.2001.918971.

[10] Crispin Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *7th USENIX Security Symposium (USENIX Security 98)*. San Antonio, TX: USENIX Association, Jan. 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[11] *Data Execution Prevention*. URL: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[12] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks". In: *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 49–54. ISBN: 9781605587882. DOI: 10.1145/1655108.1655117. URL: https://doi.org/10.1145/1655108.1655117.

[13] 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. Hong Kong, China: Association for Computing Machinery, 2011, pp. 40–51. ISBN: 9781450305648. DOI: 10.1145/1966913.1966920. URL: https://doi.org/10.1145/1966913.1966920.

[14] Yaniv David, Uri Alon, and Eran Yahav. "Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428293. URL: https://doi.org/10.1145/3428293.

[15] Ren Ding et al. "Efficient Protection of Path-sensitive Control Security". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 131–148. ISBN: 978-1-931971-40-9. URL: http://dl.acm.org/citation.cfm?id=3241189.3241201.

[16] *DynamoRio*. URL: https://dynamorio.org/.

[17] *Examining Pointer Authentication on the iPhone XS*. URL: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[18] Dan Goodin. *Apple quicktime backdoor creates code-execution peril*. URL: http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/.

[19] Mariano Graziano, D. 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* (2016).

[20] Laune C. Harris and Barton P. Miller. "Practical Analysis of Stripped Binary Code". In: *SIGARCH Comput. Archit. News* 33.5 (Dec. 2005), pp. 63–68. ISSN: 0163-5964. DOI: 10.1145/1127577.1127590. URL: https://doi.org/10.1145/1127577.1127590.

[21] Hong Hu et al. "Enforcing Unique Code Target Property for Control-Flow Integrity". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, 2018, pp. 1470–1486. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243797. URL: http://doi.acm.org/10.1145/3243734.3243797.

[22] Seunghoon Jeong et al. "A CFI Countermeasure Against GOT Overwrite Attacks". In: *IEEE Access* 8 (2020), pp. 36267–36280. DOI: 10.1109/ACCESS.2020.2975037.

[23] Mehmet Kayaalp et al. "SCRAP: Architecture for signature-based protection from Code Reuse Attacks". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 258–269. DOI: 10.1109/HPCA.2013.6522324.

[24] Jinku Li et al. "Defeating Return-Oriented Rootkits with "Return-Less" Kernels". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: Association for Computing Machinery, 2010, pp. 195–208. ISBN: 9781605585772. DOI: 10.1145/1755913.1755934. URL: https://doi.org/10.1145/1755913.1755934.

[25] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340. DOI: 10.1145/1064978.1065034. URL: https://doi.org/10.1145/1064978.1065034.

[26] Marion Marschalek. *Dig deeper into the ie vulnerability (cve-2014-1776) exploit*. URL: https://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/.

[27] Boyan Milanov. *ROPium*. URL: https://github.com/Boyan-MILANOV/ropium.

[28] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack Magazine* 49(14) (1996).

[29] *PinTool*. URL: https://software.intel.com/sites/landingpage/pintool/docs/98425/Pin/html/index.html.

[30] *pwntools*. URL: https://github.com/Gallopsled/pwntools.

[31] Jonathan Salwan. *ROPgadget - Gadgets finder and auto-roper*. URL: http://shell-storm.org/project/ROPgadget/.

[32] Sascha Schirra. *ROPper*. URL: https://github.com/sashs/Ropper/blob/master/AUTHORS.

[33] 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. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[34] Lu Si et al. "ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications". In: *Security, Privacy, and Anonymity in Computation, Communication, and Storage*. Ed. by Guojun Wang et al. Cham: Springer International Publishing, 2016, pp. 131–144. ISBN: 978-3-319-49148-6.

[35] Lu Si et al. "ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications". In: ().

[36] Blaine Stancill et al. "Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 62–81. ISBN: 978-3-642-41284-4.

[37] *The latest Adobe exploit and session upgrading*. URL: http://bugix-security.blogspot.com/2010/03/adobe-pdf-libtiff-working-exploitcve.html.

[38] Victor van der Veen et al. "Memory Errors: The Past, the Present, and the Future". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106. ISBN: 978-3-642-33338-5.

[39] Sebastian Vogl et al. "Persistent Data-only Malware: Function Hooks without Code". In: *NDSS*. 2014.

[40] Ken Westin. *GnuTLS crypto library vulnerability CVE-2014-3466*. URL: http://www.tripwire.com/state-of-security/latest-security-news/gnutlscrypto-library-vulnerability-cve-2014-3466/.