ROP chain generation: a semantic approachs

João Vasco Costa Franco
joao.vasco.franco@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2019

Abstract

Memory corruption vulnerabilities are still present in software today. While exploit mitigation techniques like Data Execution Prevention (DEP) prevent code injection, and Address Space Layout Randomization (ASLR), which randomizes the base address of memory regions, have made their exploitation very challenging, it is still feasible.

Return Oriented Programming (ROP) is the exploitation technique of choice for all modern exploits, because it requires no code injection and can therefore bypass most of the existing security protections including DEP, and has even been shown powerful in bypassing ASLR. ROP is based on reusing small pieces of code already present in the application that terminate in a `ret` instruction and chaining them together in what is called a rop-chain to achieve code execution.

However, the development of rop-chains by hand can sometimes be a difficult and slow process, due to restrictions in the payload (e.g. no zero bytes; only ASCII characters), a low amount of valuable gadgets available in the binary, etc... and so there are tools that attempt to do this automatically. Most of these tools are capable of extracting gadgets and building a rop-chain for easier to exploit binaries, but fall short on harder examples.

In this thesis we propose to evaluate the best existing rop-chain generation tools and develop a new solution that improves on them. This solution, will work by lifting all supported assembly languages to a common intermediate representation, extracting and classifying the gadgets and then, using semantic searches with an SMT solver, try to find gadgets that fulfil predetermined or user supplied exploitation recipe steps.

Keywords: Security, Binary Exploitation, Return Oriented Programming, rop-chain

1. Introduction

Memory corruption was first mentioned in an United States Air Force study on computer security requirements [2] in 1972, where it reads: "By supplying addresses outside the space allocated to the users programs, it is often possible to get the monitor to obtain unauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash."

A few years later, in 1988, the Moris Worm took the internet by storm when it spread to a believed 10% of all existing computers (around 6000 computers) becoming the first well-known buffer-overflow exploit. In the years to come, other infamous worms appeared, including Code Red in 2001 and SQL Slammer in 2003 with damages estimated in 2.6 billion dollars and 1.1 billion dollars respectively as well as many others.

This type of vulnerability is still present in software today. Mostly for performance reasons 'unsafe' programming languages like C and C++ in which programmers can directly access and change memory are still highly used everywhere, including in the operating system kernel, internet browsers, PDF readers, hypervisors, and also embedded systems where the computer power is much lower and so performance is a major priority. This means that, if by mistake, a programmer reads 64 bytes from user input to a 32 byte buffer causing a buffer-overflow, there is the potential for memory corruption which can be exploitable by an attacker. Exploitation is largely helped by the fact that the return address of a function (which redirects the application's control flow) is stored on the stack where other data lives.

Before exploit mitigations were introduced, exploitation was usually achieved by writing shellcode on the stack and then overwriting the return address with the shellcode address as shown in Figure 1, effectively achieving code execution. The shellcode can be as simple as calling an execve syscall with "/bin/sh" as the first argument, obtaining a shell on the system running the vulnerable binary.

With the increased number of vulnerabilities being exploited in the wild, causing enormous damages, it was clear that exploit mitigations were nec-

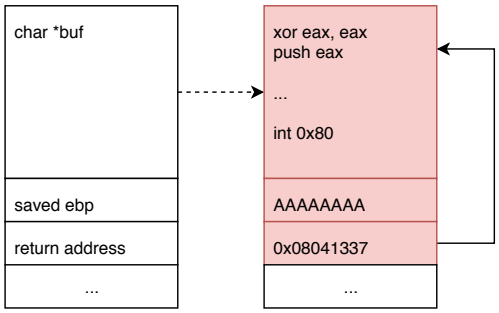

Figure 1: Shellcode Injection Example

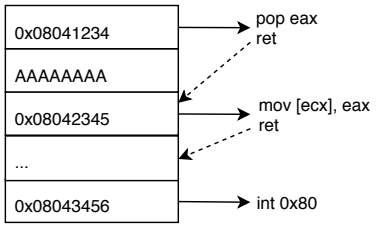

Figure 2: ROP attack example

essary. In 2004, Data Execution Prevention (DEP) [7] was introduced in Linux and Windows and later in 2006 in MacOS. DEP, also known as NX, ensures every page of memory of the program is either writable (RW-) or executable (R-X). Code segments are marked as readable and executable (R-X), while other segments such as the stack, heap, and all other data pages are marked readable and writable (RW-). This prevents the simple execution of shellcode on the stack, making the previous exploitation technique ineffective.

Another exploit mitigation technique that has made exploitation more challenging is Address Space Layout Randomisation (ASLR) [14]. It was introduced in Linux in 2005, MacOS and Windows in 2007 and in iOS in 2011. With ASLR, memory segments are allocated at a random location, and therefore an attacker is unable to know exactly where everything is laid out in memory. Since our shellcode is on the stack and the stack's address is randomized, we are not able to jump to it, which makes our previous exploit useless.

To bypass these mitigations a new technique called Return Oriented Programming (ROP) was introduced, based on re-using existing pieces of code from the application's code segment. This differs from the previous approach since the attacker does not need to write his own shellcode, but rather reuses pieces of existing code to construct his own. These pieces of code, also known as gadgets, can be chained together to create a rop-chain. Figure 2 shows as example of a rop-chain where the numbers (starting with 0x) are the addresses of gadgets from the application's binary and 'AAAAAAAA' represents data we placed on the stack. In this case, the data will end up in the eax register, because the `pop eax` instruction takes an element from the stack, puts it in eax and increments the stack pointer.

ROP is a very powerful technique which can perform arbitrary computation, but when all mitigations are present and when there is a small amount of gadgets to construct the rop-chain, it can be a very difficult and challenging task. For this reason several tools already exist that attempt to create the chain automatically, including ROPgadget [10], Ropper [11], angrop [9] and others, but they all have several flaws. The biggest one is that, for the majority of tools, the gadget search is done syntactically, i.e., with string comparison and regex searches on the assembly, instead of a semantic search. Semantic searching enables us to find gadgets based on what they do, instead of looking for specific hardcoded gadgets, which allows us to find equivalent gadgets that at first do not look very similar (e.g.: `mov rax, rbx | ret` and `xor rax, rax | not rax | and rax, rbx | ret` are equivalent).

The **goal** of this thesis is create a tool that improves on the existing ones. Through the use of an intermediate language (IR) to represent all assembly languages in a common representation and the use of a satisfiability modulo theories (SMT) solver to semantically search for gadgets, we aim to create a tool that generates chains with few gadgets, supports multiple archs and binary formats, implements sophisticated exploitation recipes, makes it easy to extend the existing exploitation recipes, including for new architectures and allows a user to create there own recipe for the problem at hand.

This tool will be helpful to ease the exploit development for security researches, aiming to receive bug bounties in competitions such as pwn2own [6]. In these competitions participants are rewarded with bounties ranging from 35'000$ to 250'000$ for finding and fully exploiting vulnerabilities in widely used software including web browsers, pdf readers, virtualizers and more. It will also be helpful to players of Capture The Flag (CTF) competitions. CTFs are cyber security competitions where teams from around the world compete against each other by solving tasks in several categories, including binary exploitation where ROP is a commonly used exploitation technique.

The remainder of this document is organized as follows. Section 2.1 will describe the existing mitigation techniques that make exploitation more challenging. Section 2.2 will explain the assumptions we make about the attacker and describe the exploitation techniques we want to implement in our tool. In Section 2.3 we will evaluate the existing

tools, denoting their weaknesses and strengths. Finally, Section 3 presents our solution and Section 4 presents the evaluation results of the best rop-chain generation tools in small, mid sized and large application.

2. Background

2.1. Mitigations

With the years and the increasing number of memory corruption vulnerabilities discovered, mitigations were developed in an attempt to prevent the bugs from being exploited. The need for this techniques comes from the fact that unsafe languages such as C and C++ are still widely used to develop applications, mostly for their efficiency benefits. For this reason, when introducing new mitigation techniques a balance between security and performance needs to be struck. If mitigation techniques harm performance too badly they will likely never see wide acceptance even if they greatly improve security.

Our tool needs to take into consideration what mitigation techniques are being applied to the application under exploitation. We will consider the following: Data Execution Prevention (DEP/NX) [7], Address Space Layout Randomization (ASLR) [14], Stack Canaries, Relocation Read-only (RELRO) and Control Flow Integrity (CFI) [1].

DEP prevents an attacker from using stack shellcode injection as an exploitation method, but making segments either writable or executable, but never both at the same time. ASLR randomizes the base addresses of the stack, heap and all libraries, however the binary itself may or may not be randomized depending on how it was compiled. Stack canaries prevent Stack-based Buffer Overflows from being exploitable, by inserting a random value between the local variables and the return address which are stored on the stack. Before returning from the function it ensures this value has no changed, and crashes otherwise. RELRO has two levels Partial-RELRO and Full-RELRO. With Partial-Relro all sections involved in symbol resolution are marked as read-only right after being initialized, except for the .got section which is still updated during runtime. On the other hand, Full-Relro resolves all symbols at startup, initializing the.got section and then marking it as read-only. Finally, CFI enforces control-flow integrity, making ROP much harder to achieve.

To have a better understanding of what mitigations are actually used in practice, we analyzed every binary installed by default on a fresh install of a Ubuntu 16.04 machine in the '/bin', '/sbin', '/usr/bin' and '/usr/sbin' directories. We observed that every binary has DEP, as well as Partial-RELRO, and that only about 20% have PIE and Full-RELRO as shown is Table 1.

Mitigation	Result	Percentage
DEP	1417/1417	100.0%
PIE	300/1417	21.2%
Canary	1183/1417	83.5%
Partial-RELRO	1416/1417	99.9%
Full-RELRO	305/1417	21.5%
CFI	0/1417	0.0%

Table 1: Mitigation usage on Ubuntu 16.04 for the default binaries

2.2. Exploitation techniques

ROP is a general technique involving the reuse of small pieces of assembly existent in the binary which terminate in a `ret` instruction called gadgets, put together in a sequence denominated a rop-chain, to achieve code execution. Even though we are already executing code inside the target application, it does not mean we have been successful in exploiting it yet. The exploit only succeeds when we are able to run arbitrary commands on the system which is running the vulnerable application. On linux systems, this is usually done by calling the libc function `system("sh")` or by invoking the syscall `execve("/bin/sh", 0, 0)` which creates an interactive shell on the targeted system.

The aim of our tool is to transform the ability to run gadgets into a successful exploit, and so we are going to explore the state of the art techniques and under what assumptions do they work.

For the rest of this document when discussing exploitation techniques, we assume that an attacker has a known exploit that allows him the control of the stack contents, including the return address and a variable length of bytes after the return address. This control can be obtained with a stack overflow or any other vulnerability that allows it. We assume that DEP is enabled, meaning no page is both executable and writable. We assume that PIE is disabled or has been previously bypassed. We assume ASLR, canaries, Partial-RELRO and Full-RELRO can be present or not. We also assume CFI is not enabled like in most common case in linux binaries.

The ROP techniques implemented in our solution will be developed for Linux ELF binaries for the x86 and x64 architectures, and, for the first release, we will only implement a subset of the ones presented in chapter 2.2. Nonetheless, our tool will support gadget extraction and searching for several other architectures, and the implementation of new techniques will be made easy, so extending it will not be hard.

2.3. ROP exploitation tools

There are several tools designed to help ROP exploitation development. All of them start by extracting the application's gadgets and some of them automatically attempt to create a full rop-chain. Between the existing ones we decided to analyze

ROPgadget [10] for its popularity and simplicity of use, Ropper [11] for its interesting semantic searcher and chain builder, angrop [9] for its use of angr's [13] powerful symbolic execution engine and finally optirop [8] for its interesting use of an intermediate language and SMT solver to do semantic searches on gadgets. One common shortcoming across all tools, is the lack of exploitation techniques implemented, which could be possible in different situations according to the gadgets available.

3. Solution

The goal of this thesis is to develop a tool that is able to, given the attack model defined in section 2.2, exploit a vulnerable application by automatically generating a rop-chain given a binary. There are several steps required to achieve this, and in the following sub-sections we will start by describing a general overview of how the tool works and then go in depth on each of its steps.

3.1. Overview

The first step is performed by the **Binary Loader** to obtain the executable segments where the gadgets reside. At this point we have the raw data with machine code of a specific architecture (e.g. ARM assembly, x64 assembly, MIPS...). If we extract and analyze the gadgets in this form, we would have to have different analysis modules for each different architecture, which is not practical.

For this reason, in the **Gadget Extractor** module, we start by lifting the specific assembly language to a common intermediate representation abstracting the specifics of each architecture, leaving the lifting itself as the only architecture dependent part of the tool. Having the machine code represented in this architecture agnostic representation, we can now extract and classify the gadgets.

Before we go into how the other steps works, we first need to understand how to use Satisfiable Modulo Theories (SMT) solvers. SMT solvers are programs that given a formula and a set of variables, attempt to find a solution, i.e., any values for the variables that satisfy the formula. For example, if we want to solve the formula '`x + 2y = 10`' we can add this formula in an SMT solver which would output one of multiple valid solutions (e.g. x=10, y=0). The specific SMT solver we will be using is called z3 [3], which is developed by Microsoft Research and has python bindings. In our use case, we need to represent registers and memory as variables with a fixed bit width, and that behave as one would expect: they overflow and support bitwise operations such as bit-shifts. Variables of this type are supported in z3 with Bit Vectors.

In the **Gadget Analysis** step, we will classify the gadgets extracted in the previous step in several ways: which registers are changed; whether the gadget writes to memory; if it makes a syscall; the gadgets z3 formula representation and the dependencies between the gadget registers. Now we are ready to match the gadgets against exploitation recipes to finally generate a rop chain.

Exploit recipes are a sequence of steps that represent exploitation techniques. To solve a recipe and generate a gadget chain, we need to match every single step with one gadget or a sequence of gadgets that accomplish that step. Furthermore we need to order the steps in such a way that they do not destroy each other. Joining all these gadgets gives us the final rop chain.

To match a single step against a gadget, we use z3 to check if there is a solution for the step, given the gadget, by adding the z3 formula representation of both the gadget and the step together. If the formula is solvable we know the gadget *can* solve the step. Consider an example where the step is '`rax = 10`' and the gadget is '`mov rax, 10 | ret`'. We would have '`And(rax_1 == 10, rax_1 == 10)`' as the combined formula, therefore z3 would say this formula is SAT. On the other hand, for the same step but for the gadget '`mov rax, 5 | ret`' the constraints would be '`And(rax_1 == 10, rax_1 == 5)`', which is obviously UNSAT, since rax_1 cannot both be 10 and 5 at the same time. However, if the formula is SAT it does not mean that the gadget solves the step, but rather that there is an input register/memory state for which the gadget solves it. From the resulting z3 model we determine which input variables are important, i.e., the input variables that the our step register depends on. If there are no important input variables, then the gadget satisfies the step completely on its own (the example above), otherwise we emit new recipe steps that represent these restrictions. By doing this we can use multiple gadgets to solve a single step. As an example, consider we have the gadget '`add rax, 10 | ret`' and our recipe step is '`rax = 10`', this would generate the new step '`rax = 0`', meaning that if rax equals 0 before the gadget, it solves the recipe step (0+10=10). We keep on using this strategy on the new steps until we succeed or until we fail by finding a UNSAT branch.

To construct a rop-chain for a given binary, we iterate through all exploitation recipes we know and try to match them with the gadgets we have. If we do not find a solution with one recipe we try the next one, until we either find a solution or have no more recipes to try.

The tool was developed in python and the modules **Binary Loader** and **Gadget Extractor** are completely modular and can be seamlessly swapped. We decided to make them modular, be-

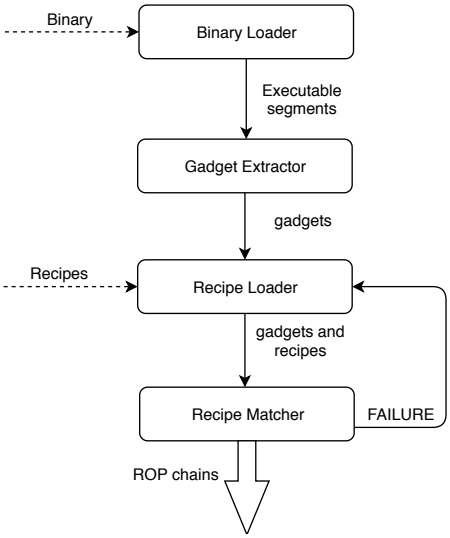

Figure 3: Solution overview

cause we used Binary Ninja, a paid (but cheap) tool, for both these modules, and this way it is possible for someone to take advantage of the core of the tool, the **Recipe Matcher**, by simply changing the loader and gadget extractor.

3.2. Binary Loader

As mentioned in the overview the Binary Loader is completely modular. We decided to implement it using Binary Ninja for simplicity, but it would be simple to implement it in any other technology. This module is responsible for loading the binary (ELF, PE, Mach-O, etc..) and extracting information we need for later stages. This information includes the executable segments (where we will look for the gadgets), the stack pointer register, the address size, and a symbol parser to help with recipe parsing.

Regarding the executable segments we align them to the default page size, 4Kb. Even if a section is describe as read only `r--` (like the `.rodata` section), what really matters is the segment in which they are mapped. If they are inside a read and execute segment `r-x` (segment $\neq$ section), they will have those permissions regardless of what the section description in the ELF defines. For this reason, the `.rodata` section is `r-x` if you compile a binary with gcc on Linux and not `r--`.

Another interesting part of the binary loader is the symbol parser, which translates symbols to addresses. If in the recipe we would like to call `system`, the symbol parser will translate the `system` string into the correct address (if the symbol exists).

3.3. Gadget Extractor

This step can be divided into 3 separate stages: machine code lifting, gadget extraction and gadget analysis.

Machine code Lifting If we aim to support multiple architectures, which we do, it is essential to lift all the different architectures (x86, x64, MIPS, arm, etc...) into a common intermediate language. This will enable us to extract and classify the gadgets, in an architecture agnostic way. There are several intermediate languages already implemented and well tested, therefore, instead of creating our own, we will use one of the existing ones. We have several options to choose from including LLVM IR [5], VEX IR [17], REIL [4] and LLIL [15].

There are three main factors we are looking for in the Intermediate Representation (IR) we will choose: 1) the existence of an api to lift from assembly to the IR; 2) this lifter has to implement as many architectures as possible; 3) it need to have a Single Static Assignment (SSA) representation, which we require when translating from IR to its SMT formula as we will discuss ahead.

In SSA form each variable is assigned exactly once, therefore variables that are assigned twice in the original code, get a new version (as if they were a new variable). Now that we know what we are looking for, we need to choose the one which best suits our needs. All of the enumerated IRs are based on SSA, leaving the lifting and supported architectures as our two main points of choice.

The LLVM IR is the base of the very popular LLVM compiler framework. Its main problem is that there is no easy way to lift the assembly to its corresponding IR. There is however, a tool called mcsema [16] which is able to do the lift assembly to its LLVM IR, but unfortunately it only fully supports the x86 and x64 architectures (with initial support for ARM).

On the other hand VEX IR, valgrind's intermediate language, supports many architectures and we get the lifting done for free using its python bindings [12]. Other tools like angr [13] and Ropper [11] have also successfully used VEX as their intermediate representation.

Finally we decided to choose, Binary Ninja's IR (LLIL), because it has everything we are looking for, including support for x86, x64, ARM, MIPS, PowerPC and more. The only disadvantage, is that it requires the user to own Binary Ninja, a paid reverse engineering platform, meaning users would have to buy Binary Ninja to be able to use our tool. This is undesirable, but since the core of the tool does not actually dependent on the Gadget Extraction and another implementation can replace it, we accept this disadvantages for the power of LLIL.

Binary Ninja has a simple API that receives an address and the raw machine code bytes, and returns the instruction represented in LLIL, which is exactly what we require for the next step.

Gadget Extraction With the executable segments where the machine code resides and a way to lift that machine code to LLIL we are ready to extract the gadgets. The algorithm to extract the gadgets, decodes an instruction for every address one by one looking for *end-instructions*, one that enables us to regain the programs control flow after being executed. This is usually a `ret` instruction, but can also be an indirect jump (`jmp rax`), an indirect call (`call rax`) or a syscall (`syscall` or `int 0x80`). When an end-instruction is found, we walk backwards from it until a max depth and, when no invalid instruction are found between the current one and the end-instruction, we store the gadget for later analysis.

We need to decoded an instruction for every address, because, even though instructions can have several bytes of length (in `x86` 1 to 15 bytes), even if we find a 2 byte length instruction we have to decode the middle of the instruction, since in `x86` and other variable length instruction architectures, if we jump to the middle of an instruction we will get a different one decoded (which is often valid) that was never intended to be there in the first place. For example, `\x41\x5f\xc3` decodes to `pop r15 | ret` (the intended instruction) but taking the first byte out, i.e., `\x5f\xc3` gives us `pop rdi | ret` which is a very useful gadget that was not in the original application, but, due to variable length instruction property, we are able to use in our ROP. Other architectures like ARM however, have a fixed size instruction length and this does not apply.

Gadget Analysis At this point we need to classify a gadget just enough to determine if it will be useful for a given recipe step. We choose to go through the LLIL, and store the following properties: which registers are changed (e.g. `(rax, rbx)`, if memory is changed (e.g. `True`), if it makes a syscall.

As an example, the gadget `mov rax, rcx | mov [rbx], 5 | ret` will be lifted to `rax_1 = rcx_0 | [rbx_0] = 5 | ret` and our classifier will store: 1) `(rax)`; 2) `True`; 3) `False`.

With this information we can decide whether it makes sense to look at a gadget for a given step. Say the step is `rax = 10`, we will only choose to match against the gadgets that have `rax` in their changed registers, which massively improves performance.

When we want to actually match against this gadget we need to do some further analysis. In this case we need to store the z3 representation of the gadgets and some helper information needed to enable the match against the recipe step. To create the z3 representation we will walk the expression tree representation of LLIL instructions. For our interests, the root nodes of an instruction are either a `LLIL_SET_REG_SSA` for setting a register or a `LLIL_STORE_SSA` for writing memory.

To get our z3 formula for `LLIL_SET_REG_SSA` nodes we walk this tree, and create a BitVec for each register and version pair, and add the formula that represents the relationships between these variables. We also store the dependencies between registers which will be useful in will in the Recipe Matcher module. As an example, for gadget `lea rax, [rax + rcx * 4]` we get the formula `rax_1 = rax_0 + (rcx_0 << 2)` and the dependencies of `rax_1` would be `rax_0` and `rcx_0`.

For `LLIL_STORE_SSA` nodes we need a way to represent memory. We accomplish this by storing a map of BitVec to BitVec, where the key is the address and the value is the memory content. For both of these we also store dependencies like with registers.

Finally with the register bit vectors, the memory map, the dependencies of each of those and the gadget constraints, we have all we need for the recipe matching step.

3.4. Recipe Loader

Before understanding **Recipe Matcher** module, we must first understand exploitation recipes. Some tools like ROPgadget define how their chain is constructed by specifying which gadgets need to be exist to complete each step. For example, ROPgadget has hardcoded that to set `rax=10` it must find the gadget `xor rax, rax | ret` to set `rax` to 0 followed by `inc rax` 10 times. While this works for large application containing many gadgets, it generates very large chains and very often, it cannot even generate the chain. In this tool we take a different approach, because our recipes do not define how `rax` is going to be set, but simply that it needs to be set to 10. The way it is going to became 10 depends on the available gadgets.

To define the steps that need to be taken to fully exploit an application, we are going to use what we called exploitation recipes. In chapter 2.2 we defined the state of the art exploitation techniques for linux ELF binaries and we implement two of these in our tool: a pure ROP exploitation recipe shown in listing 1 and a return-to-libc recipe shown in listing 2.

The constants we see in these examples (BSS, system) are resolved by the loaded binary. If the symbol cannot be resolved (e.g. system is not a valid symbol in this binary) then the recipe parsing fails.

The recipe steps can be of the following types: `SYSCALL`, `SET_REG`, `SET_MEM`, `ADD_TO_REG`, `ADD_TO_MEM`, `CALL` and `BARRIER`. The first 5 are self explanatory. A `CALL` step places the call argument

```
# Puts '/bin/sh\x00' in the BSS
[BSS] = 0x0068732f6e69622f
---
rdi = BSS
rsi = 0
rdx = 0
rax = 59
---
syscall
```

Listing 1: Pure ROP exploitation recipe example

```
# Puts 'sh\x00' in the BSS
[BSS] = 0x006873
---
rdi = BSS
---
call system
```

Listing 2: ret-to-libc exploitation recipe example

in the rop chain, i.e., if we call `system`, its address is placed in the chain. Lastly the `BARRIER` recipe step type defines a group of steps which cannot destroy the effects of the other steps and where the order of the steps is irrelevant. For example in listing 1, we put `/bin/sh\x00` in the BSS, setup the registers for the `execve` syscall and make a syscall. When setting up the registers we just care about their values before the syscall and not the order in which they got there, and therefore we place them inside a barrier.

3.5. Recipe Matcher

In this module we want to match an exploitation recipe against the gadgets we extracted and classified. Firstly we will look at how we match a single step, and then how we use it in the bigger picture to match a full recipe, guaranteeing that barriers are respected.

To match a single step, we iterate over all gadgets which are relevant for that step and use both the gadget and step z3 formulas to try and find a solution. Relevant gadgets are those that have a chance to solve the step based on the basic analysis mentioned above. For a `SYSCALL` step we look for those who have the property 'makes a syscall' set to True. For register changes, i.e., steps `SET_REG` and `ADD_TO_REG`, we choose the gadgets that have the register we want to change in the property 'registers changed'. Finally, for memory changes, i.e., steps `SET_MEM` and `ADD_TO_MEM`, we choose the gadgets that have property 'memory is changed' set to True. By doing this initial pre-selection we improve performance substantially.

To actually match the step against a specific gadget we start by ensuring the gadget is fully analyzed. Now we add its z3 formula and the step's formula to the solver. We know that SSA variables with version 0 (before any write) are input variables and the ones with the largest version (the last write) represent output variables. To create the z3 formula for steps `SET_REG` and `ADD_TO_REG`, we have to add a constraint on the last version of the destination register, i.e., the output value for this register, which we will from now one represent as `reg_X`. For `SET_REG`, the constraint will be `reg_X == 2` and for `ADD_TO_REG` it is `reg_X = reg_0 + 2`. For the steps `SET_MEM` and `ADD_TO_MEM` we constrain both the address variable and its memory content. For `SET_MEM` the constraint would be And(mem_addr_i == 0x400000, mem_content_i == 2) and for `ADD_TO_MEM` it would be And(mem_addr_i == 0x400000, mem_contents_i == [mem_addr_i] + 2).

Now the solver will either output SAT, if there was a solution, or UNSAT otherwise. If the formula is SAT we know the gadget *can* solve the step. Consider an example where the step is `rax = 10` and the gadget is `mov rax, 10 | ret`. We would have `And(rax_1 == 10, rax_1 == 10)` as the combined formula, therefore z3 would say this formula is SAT. On the other hand, for the same step but for the gadget `mov rax, 5 | ret` the constraints would be `And(rax_1 == 10, rax_1 == 5)`, which is obviously UNSAT, since `rax_1` cannot both be 10 and 5 at the same time.

However, even if the formula is SAT, it does not mean that the gadget solves the step, but rather that there is an input register/memory state that solves it. Z3 provides us with the values of every variable for the solution if found, but not all of these are important. To start we are only interested in the variables which have an SSA version of 0, i.e., the input variables. However, even these input variables may not be important. Consider the example gadget `rax_1 = 5 | rbx_1 = rbx_0 + 5 | ret` and the step `rax = 5`. `rbx_0` is the only input register, but is has no influence on `rax_1`, and so it is not important, which we can determine by looking at the dependencies of `rax_1` we calculated during our analysis.

If we determine that there are no important input variables, then the gadget satisfies the step completely on its own (the example above), otherwise we emit new recipe steps that represent these restrictions, chaining multiple gadgets to solve a single step. As an example, consider we have the gadget `rax_1 = rax_0 + 10 | ret` and the recipe step is `rax = 10`. `rax_0` is the only important variable and so, we would generate the new step `rax = 0`, meaning that if `rax` equals 0 before the gadget, it solves the recipe step. We keep on using this strategy on the new steps until we succeed or until we fail, by finding a UNSAT branch.

For the `ADD_TO_REG` step we have to be specially careful, because z3 will simply give us one solution and not prove that it works for any input value. Therefore for step `rbx += 2` the lifted gadget `rax_1 = 2 | ret` would have as important vars `rax_0 = 0`, because indeed, if `rax` was 0 and we set it to 2 we have added 2 to `rax`. Nonetheless this is not what we are looking for, and so, as an heuristic, we constraint the input register to 3 different values (e.g. `rax_0 == 1337` and `rax_0 == 0xdeadbeef`) and only if all of them succeed do we mark the gadget as good.

Besides the specified step restrictions we also ensure that memory writes are within a writable memory region, to avoid our chain from accessing invalid memory addresses and crashing. We encode this in a z3 formula and get the necessary register sets as important vars.

We represent the final result of matching all steps against the available gadgets in a tree with three types of nodes: *steps* nodes, *single step* nodes and *gadget nodes*. Firstly we have a *steps* node, which represents a collection of steps that *all* have to be accomplished, regardless of the order they are executed in. These nodes perfectly encode the barrier properties. The *steps* nodes have a list of *single-step* nodes as children. These represent a single exploitation step and can have as children a list of *gadget* nodes and *steps* nodes. These *steps* nodes contain the new steps generated while matching the gadget and the solver found a solution but there were important variables, as was explained above. Lastly, a *gadget* node represents a simple gadget that solves the given step, and can never have children. A *single-step* node can have several of these children nodes but only one of them need to be accomplished, i.e., if we have multiple alternative gadgets or steps, one of them suffices.

Figure 4 shows one of these trees for the step `rax = 2` and gadgets `mov rbx 2 | ret` and `mov rax, rbx | ret`. Since the architecture used is x64, other instructions in the middle of the intended instructions are also found.

To extract all possible chains we walk the tree and find all paths from root steps node to gadget nodes, by forking the chain on the children of *single-step* nodes and by joining the children of *steps* nodes. For the tree in figure 4 we extract the chains `mov rbx, 0x2 | ret | mov rax, rbx | ret` and `mov rbx, 0x2 | ret | mov eax, ebx | ret`.

While joining the children of a *steps* nodes, we need to find a valid order for these steps if any exists. Determining the order by which we set the registers is a problem that can be modeled as a directed graph where the nodes are the registers and the edges represent a 'destroys' relation. If the graph has a cycle there is no possible ordering, because it means that setting R1 destroys R2 and to set R2 destroys R1 (can be generalized to any length of cycle), meaning we can never have both registers set at the same time without destroying the other. On the other hand, if we do not have a cycle we can construct our register setting order by: 1) finding the register which has no edges pointing to it (always exists in an acyclic graph), set that register and remove it from the graph; 2) repeat step 1 until there are no nodes left. Intuitively this method always sets the register no other registers will destroy (i.e. has no edges pointing to it) until all registers are set.

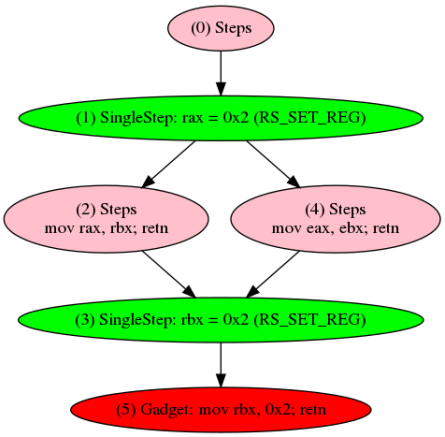

Figure 4: Result tree example

4. Evaluation

In order to determine the success of our tool, we have to compare it against the best existing ones. In this section we are going to describe the setup we used, the metrics of the evaluation and the results.

4.1. Setup

To have a complete view of the tools, we are going to test them against a set of varied binaries. These applications will come from: 1) a set of specially built binaries with very few gadgets and a specific recipe, to test the limits of what the tools are capable of in terms of matching; 2) a mid size application: `/bin/ls` (from Ubuntu 18.04 LTS); 3)a large Ubuntu 18.04 LTS library: `/lib/x86_64-linux-gnu/libc.so.6`.

The gadgets in 1) are associated with 4 different recipes, one for each of the main recipe steps described in section 3.4: `rax = 2` for `SET_REG`; `rbx += 2` for `ADD_TO_REG`; `[0x400000] = 0x1337` for `SET_MEM`; `[0x400000] += 2` for `ADD_TO_MEM`.

For each of these steps we have two types of tests, ones that should succeed and ones that should fail, to test if the tools are able to generate chains, but also that they do not find false positives. For example, in the `rbx += 2` step, we have a test to

Tool/Step	rax=2	rbx+=2	[mem]=0x1337	[mem]+=2
ROPGadget	N/A	N/A	N/A	N/A
ropper	4/11	0/5	N/A	N/A
angrop	1/11	N/A	0/6	0/4
our solution	10/11	3/5	4/6	3/4

Table 2: Tests that should succeed (passed/total)

Tool/Step	rax=2	rbx+=2	[mem]=0x1337	[mem]+=2
ROPGadget	N/A	N/A	N/A	N/A
ropper	6/6	6/6	N/A	N/A
angrop	6/6	N/A	3/3	2/2
our solution	6/6	6/6	3/3	2/2

Table 3: Tests that should fail (passed/total)

ensure the gadgets `mov rbx, 0 | ret` followed by `mov rbx, 2 | ret` do not work. Technically we are increasing the value of rbx by 2, but in practice this is not what we are looking for. We want to increment whatever value was already in rbx in the first place. As for the *should succeed* tests we are trying to find if the tools are capable of chaining gadget to accomplish a single step. For the `rax = 2` step we have for example the test with gadgets `mov rbx, 2 | ret` and `mov rax, rbx | ret` where the resulting chain has to set rbx to 2 and then move that value into rax with the second gadget. With all these tests we are able to check both how powerful the tools are at generating rop-chains with a low amount of gadgets, and how fast they are in large libraries like `libc`. As for the tools we are going to test against, we chose ROPgadget [10], Ropper [11] and angrop [9].

4.2. Metrics

The metrics extracted per application, for each rop-chain generation tool, are: 1) can it generate a rop-chain for the binary; 2) what is the size of the generated chain; 3) how much time it took to generate.

If the tool can generate a chain at all, we also measure how much time it took and what was its size. The lesser the size the better, since, depending on how much of the stack an attacker controls, a large chain may not be suitable.

Regarding the time, it can differ from tool to tool depending on the implementation. The amount of extracted gadgets should be similar among the tools since the algorithm is straight forward and only minor differences can exist, but a recursive implementation that walks backwards from `ret`'s may be faster than one that just goes through the whole binary linearly. Also a tool that does any sort of analysis and classification at extraction, will likely take more time to finish. Besides the algorithmic considerations, the language used (C++ vs python), if parallelism is used, etc... will also make the extraction and matching faster or slower.

5. Results

Tables 2 and 3, show the results of running the set of binaries with very few gadgets that should succeed and those that should fail respectively.

To start, ROPGadget does not support semantic search and can only generate a full `execve` chain, therefore it does not make sense to test it in this case. On the other hand ropper supports semantic searches, but only for register changes and not memory. The command used for testing was `ropper -f {test_path} --semantic "{recipe}"` where test_path is the binary path and recipe is either `rax==2` or `rbx+=2`. Ropper also uses z3 for matching the semantic search, but contrary to our tool, it is not able to join multiple gadgets to solve a step. For this reason, we can see it solved all recipes that were solvable with one gadget, but not the other ones. The only exception being `pop rax | ret`, since it does not support memory accesses. Lastly, for a reason we were unable to determine, ropper was not finding the useful gadgets of the `rbx+=2` tests, and sequentially was not able to find a chain.

In the case of angrop there is no command line utility and so we need to write a small python script to be able to run the tests. This script is shown in appendix A.

Surprisingly to us, angrop did very poorly with our tests, only passing one `rax = 2` test out of all the tests. The reason for this is that it does not find direct setting of registers with constants, i.e., it fails to find `mov rax, 2` as a gadget for setting rax. This is also the reason why `pop rax | ret` is the only gadget it was able to find a chain for. In real world binaries, pop's are the main way to set registers, so the results may not reflect the real power of angrop in this case, but nonetheless sometimes there are situations where it would not find gadgets exactly for these reasons. Finally our tool was able to pass most of these tests with only a few exceptions which we will dive into now.

For the step `rbx += 2`, it fails on gadget `inc rbx | ret`, because, as of right now, we only emit new steps as register sets (`SET_REG`) and in this case we would have to emit a `rbx += 1` (an `ADD_TO_REG`). There is no reason why we could not do it, but it is simply not implemented yet.

The test with gadgets `mov rax, rbx | ret`, `add rax, 0x2 | ret` and `mov rbx, rax | ret` also fails. The intended chain (in simple terms) is `rax = rbx | rax += 2 | rbx = rax`. The reason this also fails is that we have no way to express that after `mov rax, rbx | ret` we would need a `rax += 2` and a `rbx = rax`. We have no way to express that a register needs to be set to another register (`rbx = rax`), nor the order in which the resulting steps have to be executed (`rbx = rax` only after `rax += 2`). This is a very specific situation

Tool/Metric	chain generated	chain size	chain time
ROPGadget	NO	N/A	N/A
ropper	NO	N/A	N/A
angrop	YES	15 qwords	49 seconds
our solution	YES	20 qwords	42 seconds

Table 4: ROP tool comparison for /bin/ls

Tool/Metric	chain generated	chain size	chain time
ROPGadget	YES	76 qwords	16 seconds
ropper	YES	27 qwords	15 seconds
angrop	YES	12 qwords	187 seconds
our solution	NO	N/A	N/A

Table 5: ROP tool comparison for /lib/x86_64-linux-gnu/libc.so.6

we belief is very unlikely to ever appear in a real binary, and so we accept the failure in exchange for the simplicity of our architecture.

The next tests, shown in tables 4 and 5, show how the tools perform when generating a full rop-chain. We chose `/bin/ls` as a mid size binary (131KB) and `libc` as a large one (2MB).

To generate the chains for each tool we used the following commands and scripts: for ROPGadget `ROPgadget --binary /bin/ls --ropchain`; for ropper, `ropper --file /bin/ls --chain execve` and for angrop, with the script in listing 3 where the chain is `chain = rop.execve()`.

Looking at the case of `/bin/ls`, we can see that only our tool and angrop were able to generate this chain. Angrop slightly wins in terms of chain size, but ours wins in the time category. Both performances are practically equal, the only difference in the chain being that angrop sets rax using both these gadgets `pop rdi | ret` and `mov rax, rdi | ret` combined, while our tool does it using only one gadget pop rax | pop rbx | pop rbp | pop r12 | pop r13 | pop r14 | pop r15 | ret. Our solution uses 8 stack qwords (caused by the pops) to only 3 qwords used by angrop's solution.

In the `/lib/x86_64-linux-gnu/libc.so.6` case all tools but ours are capable of generating a chain. Time wise ropper and ROPGadget are superior, because they do not do any lifting or semantic searching, while length wise, angrop is far better, even though it takes over 3 minutes to finish.

With this test, we concluded that our tool, especially the gadget extraction step, did not scale very well with application size, most notably memory wise, since we run out of memory even before getting to the gadget matching stage. In the future work section we discuss how we can improve this lack of performance.

6. Conclusions

In conclusion, we developed a tool which security researchers and CTF players alike, will be able to use to finalize their exploit of a vulnerable application, assuming they already control the stack contents, including the return address and a variable length of bytes afterwards, and that the application can have several exploitation mitigations applied to it.

We do this by creating a rop-chain, built in 3 main steps. Firstly we load the binary to find the executable segments where we will find the gadgets. Secondly we extract these gadgets by lifting them to LLIL, Binary Ninja's intermediate representation (IR), extracting them and analyzing them. Finally, we try to match them against predefined exploitation recipes, using an SMT Solver to extract new equivalent steps, effectively chaining multiple gadgets to solve a single step.

Our tool also supports a large number of architectures, and is very effective when creating rop chains for smaller applications where interesting gadgets are scarce.

6.1. Future Work

In its current state, our tool does almost everything we want in terms of extracting gadgets and matching those gadgets against predefined or user supplied recipes, but there are improvements that can be made in terms of usability and performance.

To start, a graphical interface implemented as a Binary Ninja plugin would be a very easy way to interact with it. There would be a panel with the gadgets and a search bar to filter them, a button to run the built-in exploitation recipes and text area for user specified recipes.

In terms of performance and memory usage there are also possible improvements, as we found out during evaluation of large binaries in section 4. Right now, we extract all gadgets, and only then start to match the steps. If instead we did both steps at the same time, i.e., extract the next n gadgets, match on those, and only extract more gadgets if no chain was found, we would avoid extracting gadgets that will never actually be used. Furthermore, parallelism and implementation of the gadget extraction portion of the tool in C++ instead of python would drastically improve performance.

One other interesting feature, would be to output constrained chains when we cannot find a full one, i.e., even if some step is unmatched, output the chain coupled with a condition. For example if we wanted to match step `rax = 2`, but the only available gadget was `add rax, 2 | ret` we would output this gadget under the condition that rax was 0 just before the chain starts to execute.

Finally, more exploitation recipes need to be implemented. We also still do not take into consideration the mitigations that are being applied which is crucial to determine the exploitation techniques are going to be effective.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, Co Fort Washington PA, 1972.

[3] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis, 2009.

[5] LLVM. LLVM language reference manual. (visited on 29/05/2018).

[6] T. Micro. Pwn2Own 2018 rules, 2018. (visited on 29/05/2018).

[7] Microsoft. Understanding DEP as a mitigation technology, 2009. (visited on 29/05/2018).

[8] N. A. Quynh. OptiROP: the art of hunting ROP gadgets, 2013.

[9] salls. angrop tool, 2018. (visited on 29/05/2018).

[10] J. Salwan. Ropgadget tool, 2018. (visited on 29/05/2018).

[11] S. Schirra. Ropper tool, 2018. (visited on 29/05/2018).

[12] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Python bindings for Valgrind's VEX IR, 2015. (visited on 29/05/2018).

[13] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[14] P. Team. Pax address space layout randomization (aslr), 2003. 2003. (visited on 29/05/2018).

[15] TrailOfBits. Breaking down Binary Ninjas low level IL, 2017. (visited on 29/05/2018).

[16] TrailOfBits. Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode, 2018. (visited on 29/05/2018).

[17] Valgrind. Vex IR, an architecture-neutral intermediate representation. (visited on 29/05/2018).

A. Angr script

```python
import angr, angrop

def angrop_search(filename, recipe):
    # Load bin
    p = angr.Project(filename)
    rop = p.analyses.ROP()

    # Find the gadgets
    rop.find_gadgets()

    # Generate the chain
    try:
        if recipe == 'set_rax_to_2':
            chain = rop.set_regs(rax=2)
        elif recipe == 'add_mem':
            chain = rop.add_to_mem(0x400000, 2)
        elif recipe == 'write_mem':
            chain = rop.write_to_mem(0x400000, '\x13\x37')

        got_chain = True

    except angrop.errors.RopException:
        got_chain = False

    return got_chain
```

Listing 3: angrop test script

B. Tests

B.1. Tests for recipe rax = 2

```
{
    "success_tests": [
        [
            "mov rax, strict qword 2; ret"
        ],
        [
            "mov eax, 2; ret"
        ],
        [
            "xor rax, rax; ret",
            "mov al, 2; ret"
        ],
        [
            "pop rax; ret"
        ],
        [
            "xor rax, rax; ret",
            "inc rax; ret"
        ],
        [
            "mov rax, strict qword 0; ret",
            "inc rax; ret"
        ],
        [
            "mov rbx, strict qword 2; ret",
            "mov rax, rbx; ret"
```

```
        ],
        [
            "mov rax, strict qword 3; dec rax; ret"
        ],
        [
            "mov rbx, strict qword 0; ret",
            "mov rcx, strict qword 2; ret",
            "add rbx, rcx; mov rax, rbx; ret"
        ],
        [
            "pop rax; inc rax; ret"
        ],
        [
            "pop r12; ret",
            "mov rax, 2; mov qword [r12], 10; ret"
        ]
    ],
    "failure_tests": [
        [
            "mov rax, strict qword 0; ret"
        ],
        [
            "mov rax, strict qword 2; mov rax, strict qword 0; ret"
        ],
        [
            "mov al, 2; mov ah, 2; ret"
        ],
        [
            "mov ah, 2; ret"
        ],
        [
            "pop rax; mov qword [0xdeadbeef], 2; ret"
        ],
        [
            "pop rbx; ret"
        ]
    ]
}
```

B.2. Tests for recipe rbx += 2

```
{
    "success_tests": [
        [
            "add rbx, strict qword 0x2; ret"
        ],
        [
            "mov rax, strict qword 0x2; ret",
            "add rbx, rax; ret"
        ],
        [
            "inc rbx; ret"
        ],
        [
            "mov rcx, strict qword 0; ret",
            "mov rdx, strict qword 0x2; ret",
```

```
            "add rcx, rdx; add rbx, rcx; ret"
        ],
        [
            "mov rax, rbx; ret",
            "add rax, strict qword 0x2; ret",
            "mov rbx, rax; ret"
        ]
    ],
    "failure_tests": [
        [
            "or rbx, strict qword 0x2; ret"
        ],
        [
            "add rbx, strict qword 0x2; pop rbx; ret"
        ],
        [
            "mov rax, strict qword 0x2; ret",
            "sub rbx, rax; ret"
        ],
        [
            "mov rbx, strict qword 2; ret",
            "mov rbx, strict qword 0; ret"
        ],
        [
            "mov rcx, strict qword 0; ret",
            "mov rbx, strict qword 0x2; ret",
            "add rcx, rbx; add rbx, rcx; ret"
        ],
        [
            "pop rbx; ret"
        ]
    ]
}
```

B.3. Tests for recipe [0x400000] = 0x1337

```
{
    "success_tests": [
        [
            "mov qword [0x400000], 0x1337; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "mov qword [r12], 0x1337; ret"
        ],
        [
            "mov r12, strict qword 0x3ffff8; ret",
            "mov qword [r12 + 0x8], 0x1337; ret"
        ],
        [
            "mov rcx, strict qword 1; ret",
            "mov r12, strict qword 0x3ffff8; ret",
            "mov qword [r12 + rcx*8], 0x1337; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
```

```
            "mov qword [r12], 0x1336; ret",
            "inc qword [r12]; ret"
        ],
        [
            "pop r12; ret",
            "mov qword [r12], 0x1337; ret"
        ]

    ],
    "failure_tests": [
        [
            "mov r12, strict qword 0x400000; ret",
            "mov qword [r11], 0x1337; ret"
        ],
        [
            "mov r12, strict qword 0xcafe; ret",
            "mov qword [r12], 0x1337; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "mov qword [r12], 0x10; ret"
        ]
    ]
}
```

B.4. Tests for recipe [0x400000] += 2

```
{
    "success_tests": [
        [
            "add qword [0x400000], 2; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "add qword [r12], 2; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "inc qword [r12]; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "mov rax, strict qword 2; ret",
            "add qword [r12], rax; ret"
        ]
    ],
    "failure_tests": [
        [
            "mov qword [0x400000], 0; ret",
            "mov qword [0x400000], 2; ret"
        ],
        [
            "mov r12, strict qword 0x400000; ret",
            "add qword [r12], rax; ret"
        ]
    ]
```

```
30 }
```
