Return-Oriented Programming Attacks Modelling as Graph Traversal Problems

Billie Bhaskara Wibawa - 13524024[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]billiebaskarawibawa101@gmail.com, 13524024@std.stei.itb.ac.id

***Abstract*— Return Oriented Programming (ROP) is the act of using existing assembly instructions inside a binary executable to create a series of instructions that executes commands outside of the intended use of the original binary executable. This paper proposes a novel formalization of the ROP exploit construction process as a graph traversal problem. Specifically, this paper will model the search space for viable gadget chains as paths within a directed graph, where nodes represent states of the registers and memories and edges represent feasible transitions between them using gadgets. This enables the application of established graph-theoretic algorithms and analysis to systematically investigate the properties, complexity, and practical limitations of ROP attacks. By framing ROP exploitation in this graph-theoretic context, our approach aims to provide deeper insights into attack feasibility, gadget availability constraints, and the effectiveness of ASLR/PIE as a mitigation strategy.**

***Keywords*—ASLR, Gadget, Position Independent Executable, Return Oriented Programming**

I. Introduction

Binaries, or executables, are machine code for a computer to execute. Binary exploitation involves analyzing and manipulating executable files to uncover vulnerabilities that allow unauthorized control flow manipulation, often through reverse engineering of machine code.

A ROP-based exploit specifically is the act of repurposing existing machine code fragments already present in the program's memory to construct malicious logic. The machine code fragments that is being repurposed to construct the exploit is called gadgets. Gadgets must end with a "ret" assembly insruction as this allows the attacker to redirect code execution back to the stack in which the attacker has already prepared a series of gadgets that executes the exploit. The primary goal of a ROP-based exploit is typically to invoke the system() function from the C standard library, passing the string "/bin/sh" as its argument. This executes a shell command, granting the attacker terminal access to the vulnerable machine and potentially exposing sensitive files and system resources.

The fundamental mechanism enabling ROP gadget exploitation begins with a buffer overflow vulnerability, where an attacker supplies more input data than the program's buffer can hold. This overflow corrupts the call stack, specifically targeting the return address of the vulnerable function. By carefully crafting their input, the attacker overwrites this return address to point not to legitimate code, but rather back into the stack memory itself. When the compromised function completes and executes its ret instruction, instead of returning to the caller, the program begins executing whatever data resides at the attacker-specified stack location. The attacker then prepares the stack with a carefully constructed sequence of gadget addresses, effectively creating a malicious program by chaining together existing code fragments. Each "ret" instruction at the end of a gadget acts as a logical jump to the next gadget in the chain, allowing the attacker to stitch together complex behaviors from these borrowed code pieces.

Address Space Layout Randomization (ASLR) was developed specifically to combat this type of exploitation. ASLR is a kernel-level security technique that randomizes the memory addresses used by system components (stack, heap, shared libraries) when a process starts. It makes memory addresses non-deterministic across different runs of a program. This effectively makes it difficult for the attacker to construct the exploit due to the fact that the main mechanism of a ROP-based exploit is to redirect code execution to execute functions inside shared libraries.

Another layer of protection to combat ROP-based exploitation is the implementation of Position Independent Executables (PIE). In summary, PIE is a form of compiled binary in which the position of instructions inside the binary itself is randomized. The difference between PIE and ASLR is that PIE randomizes the location of functions inside the binary itself, such as the main function. While ASLR randomizes the functions inside the shared C standard library.

II. Binary Exploitation

Binary exploitation is the art of identifying and leveraging vulnerabilities within a compiled program, or binary, to force it to behave in unintended ways, ultimately for the purpose of compromising a system. The end goal in binary exploitation is typically to achieve arbitrary code execution, enabling the attacker to run commands, spawn a shell, and gain control over the machine with the privileges of the exploited application.

In binary exploitation, there are specific techniques that an attacker can use to reach their goal. Some techniques are dependant on other techniques such as ROP being dependant on

buffer overflow. In the following section this paper will cover each techniques in greater detail.

A. Buffer Overflow

A buffer overflow is a type of software vulnerability that occurs when a program writes more data to a buffer (a temporary storage area in memory) than it can hold. This causes the excess data to overflow into adjacent memory spaces, potentially corrupting data, crashing the program, or allowing attackers to execute malicious code.

The most common type of buffer overflow and the type that is relevant to the topic of this paper is a stack-based buffer overflow. A stack-based buffer overflow is a buffer overflow in which the data that are being overwritten reside in the stack. This is because the buffer (usually in form of a character array) is not allocated using the malloc() function in C or its equivalent. Rather, its explicitly defined with a fixed size similar to the code snippet shown below.

```
int buffer[BUFF_SIZE];
// Code 2.1
```

The vulnerability that is being exploited in a stack-based buffer overflow is the fact that the return address that is pointed by the stack pointer resides in the stack itself. This means that if we have the capability to overwrite the stack, we also have the capability to overwrite the return address pointed by the stack pointer.

Below is a figure showing the condition of the stack when executing a vulnerable function. Note that ESP or the stack pointer does not point to the return address. This is merely because the function is not yet completed.

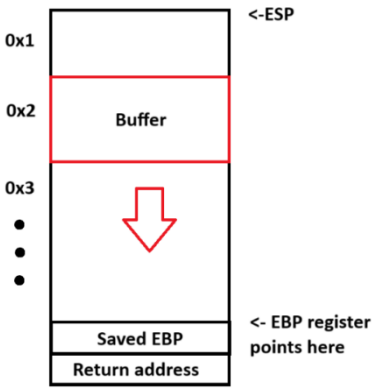


Fig 2.1 The position of the stack pointer and the base pointer in a function call

The following figure is a figure showing what the stack looks like right before the program executes a “RET” instruction.

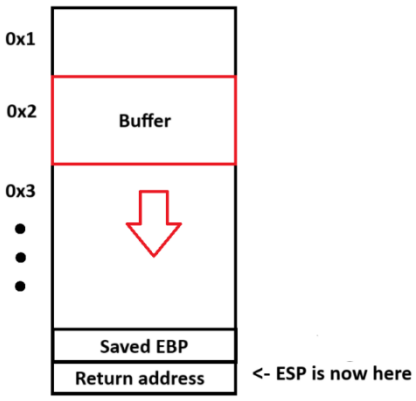


Fig 2.2 The position of the stack pointer right before the “ret” instruction

Here is a demonstration of a simple program that has a stack-based buffer overflow vulnerability.

```
void vuln(){
    char buffer[6];
    fgets(buffer,100,stdin); //takes 100
characters, larger than the buffer!
}
void win(){
    puts("You're not supposed to be here!");
}
int main(){
    vuln();
    return 0;
}
// Code 2.2 A vulnerable program
```

And here is a simple script that exploits our vulnerable program.

```
from pwn import *
p = process("./vuln") //we have compiled our
vulnerable program with the name “vuln”
win_address = p64(0x401159)
payload = b"A"*14 + win_address
p.sendline(payload)
p.interactive()
// Code 2.3 An example script to exploit a
buffer overflow
```

The python script above will overflow the buffer with a padding that consists of 14 A’s and then put the address of our win() function right after the padding. Running the Python script will make our program output the message “You’re not supposed to be here!” that originally should not be outputted since we never call the win() function. Because we overwrite the return address in the stack, the program does not comeback to main after calling vuln(), but instead goes to the win() function.

B. Return Oriented Programming

Return-Oriented Programming (ROP) stands as a sophisticated and potent technique in the arsenal of cyber-attackers, allowing them to execute malicious code on a target system despite the presence of modern security defenses. It represents a significant evolution from traditional code injection attacks by cleverly repurposing existing, legitimate code for nefarious ends. This method of attack is particularly effective against systems that employ security measures like Data Execution Prevention (DEP) or Non-executable (NX) bit, which are designed to prevent the execution of code from memory regions intended for data, such as the stack.

The core principle of ROP lies in the creative abuse of a program's control flow, typically initiated by a memory corruption vulnerability such as a stack buffer overflow. When a function is called, the address of the instruction to which it should return upon completion is stored on the call stack. An attacker can exploit a buffer overflow to overwrite this return address with the address of a chosen piece of code. However, instead of pointing to the start of a whole function, it only points to parts of this function to use only specific instructions.These small snippets are what we call as "gadgets."

The following figure shows how a buffer overflow allows an attacker to construct a gadget chain. Due to the fact that the “ret” assembly instruction will redirect code execution to the address that is pointed by the stack pointer and then pops the stack, we can chain gadgets such that after one gadget is executed it continues to the next address in the stack, which is the address of another gadget the attacker has prepared.

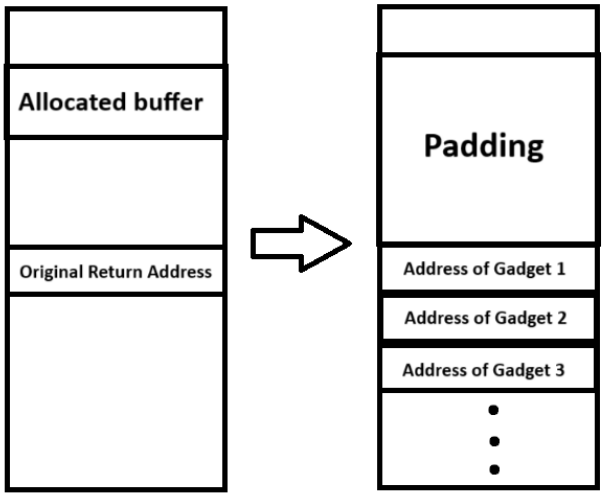


Fig 2.3 Illustration of a ROP chain in the stack

C. ASLR and PIE Bypassing

A foundational defense against Return Oriented Programming is Address Space Layout Randomization (ASLR), a technique employed by modern operating systems to thwart attacks that rely on predictable memory layouts. ASLR works by randomizing the base addresses of key memory segments—such as the stack, heap, and shared libraries—each time a program is launched. This randomization means an attacker can no longer rely on hardcoded addresses to locate functions or ROP gadgets needed for their exploit. If the attacker aims for a function in a library, its location will be different with every execution, causing a payload that works once to crash the program on the next attempt.

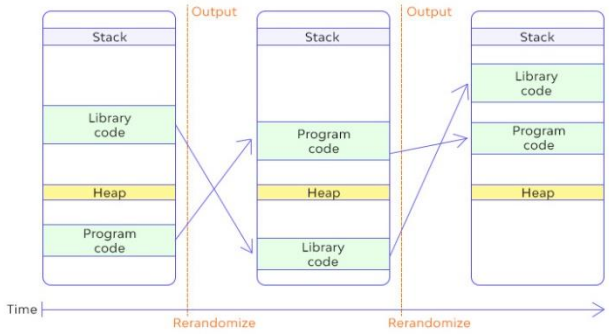


Fig 2.4 Address space layout randomization example

(Taken from wallarm.com/)

However, ASLR in its initial form had a significant limitation: while it effectively randomized libraries and other data segments, the main program executable itself was often loaded at a fixed, predictable address. This created a static island of code that attackers could reliably analyze to find gadgets for their ROP chains, undermining the protection ASLR was meant to provide. To close this loophole, the concept of Position-Independent Executables (PIE) was introduced. PIE is a compiler feature that generates binary code using relative addressing instead of absolute memory addresses. This means the executable's code does not depend on being loaded at a specific location; it can function correctly from anywhere in memory.

The true power of this defensive strategy is realized when ASLR and PIE work in synergy. When an executable is compiled as a PIE, the operating system's ASLR mechanism is empowered to load the main program at a new, random memory address upon every execution, just as it does for libraries. This eliminates the last remaining static region of code an attacker could depend on. Together, a PIE-enabled binary running with full ASLR forces an adversary into a much more difficult position. They must not only find a memory corruption bug but also a separate information disclosure vulnerability to leak the randomized addresses before they can even begin to construct a reliable exploit, significantly raising the complexity and cost of a successful attack.

Although it is difficult to bypass ASLR and PIE when they are combined, we still have several viable options to bypass security. One of the most consistent option is to try and find memory leaks in other parts of the program. This means that even with PIE and ASLR, the attacker can still calculate offsets of specific points of interest at runtime. Even without direct memory leak of a specific point of interest, the attacker can still calculate the address of these points of interest by calculating the offset between them and known leaked pointers.

III. GRAPH THEORY

In the realm of mathematics, particularly within the branch of graph theory, a graph is a fundamental structure used to model pairwise relationships between objects. It is a mathematical

representation of a network, consisting of a set of objects and the connections between them.

At its core, a graph is formally defined as an ordered pair

$$G = (V, E)$$

where V is a finite, non-empty set of vertices (also known as nodes or points), and E is a set of edges (also called lines or arcs). Each edge connects a pair of vertices. An edge represents a relationship or connection between the two vertices it links.

In the realm of mathematics, particularly within the branch of graph theory, a graph is a fundamental structure used to model pairwise relationships between objects. It is a mathematical representation of a network, consisting of a set of objects and the connections between them.

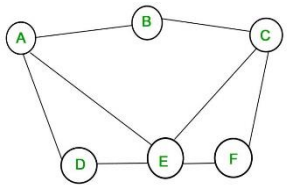


Fig 3.1 Illustration of a graph

(Taken from geeksforgeeks.org)

A. Types of Graph

Based on the organization of edges in a graph, we can classify graphs as these two categories

1. Simple graph

Fig 3.1 Simple graph

(Taken from mathworld.wolfram.com)

A simple graph is a type of graph that is undirected, contains no loops (edges connecting a vertex to itself), and has no multiple edges between any two vertices.

2. Non-simple graph

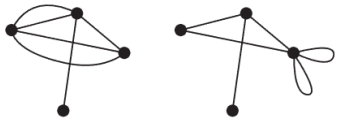


Fig 3.2 Non-simple graph

(Taken from mathworld.wolfram.com)

A non-simple graph is a graph that does not adhere to the restrictions of a simple graph. Specifically, it can contain multiple edges between the same pair of vertices (also known as parallel edges) and/or loops (edges connecting a vertex to itself).

Based on the orientation of the edges in a graph, we can classify graphs as these two other categories

1. Undirected graph

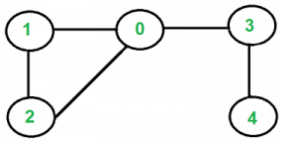


Fig 3.3 Undirected graph

(Taken from geeksforgeeks.org)

An undirected graph is a type of graph where edges have no specific direction, meaning the connection between two nodes is bidirectional. In simpler terms, if there's a path from node A to node B, there's also automatically a path from node B to node A.

2. Directed graph

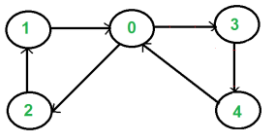


Fig 3.4 Directed graph

(Taken from geeksforgeeks.org)

A directed graph, also known as a digraph, is a graph where each edge has a direction, meaning it points from one vertex (node) to another. Unlike undirected graphs where edges connect nodes bidirectionally, directed graphs have one-way connections.

IV. ROP GRAPH MODELLING

Delving into the intricate world of Return-Oriented Programming (ROP) attacks, a powerful analytical lens can be found in the well-established field of graph theory. By modeling the components of an ROP attack as a graph, the search for a viable exploit can be transformed into a classic pathfinding problem, providing a formal and systematic way to analyze attack complexity and defense mechanisms.

A. Definition of Edges and Vertices in ROP

To formalize the construction of a Return-Oriented Programming (ROP) exploit, we propose a graph-based model where the problem is reframed as a search within a state-space. In this model, the vertices of our graph represent the achievable states of the machine by the use of gadgets. Here by state it means a specific configuration of the CPU's general-purpose registers. With states represented using vertices, it is logical to

then proceed defining egdes as the edges of the graph naturally represent the gadgets that transition the machine from one state to another. An edge from a source state S1 to a destination state S2 exists if and only if there is a gadget that, when executed from S1, results in the new state S2. This transition encapsulates the full effect of the gadget: the instructions it executes and any values it pops from the stack into registers. Each edge, therefore, acts as a directed, functional link, transforming the machine's context in a well-defined way. The properties of this transition, such as the gadget's memory address and the number of bytes it requires on the stack, can be assigned as weights to the edge, enabling quantitative analysis.

This vertex-as-state abstraction is crucial because the ultimate objective of a ROP attack is not simply to execute a sequence of instructions, but to drive the CPU into a precise final state that enables a malicious action, such as initiating a system call with specific arguments. The initial vertex in our graph represents the machine state at the moment an attacker gains control of the instruction pointer, and the goal is to find a path to a target vertex representing the desired final state.

The following figure is an illustration on how a simple ROP of which the objective is to call system("/bin/sh") on a x86-64 linux system.

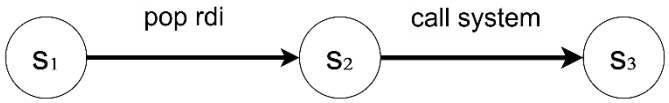


Fig 4.1 Graph of a simple ROP exploit

The S_1 represents a state in which a pointer to the string "/bin/sh" is on the stack. S_2 represents a state in which the string "/bin/sh" is now loaded on the rdi register. S_3 represents a state in which a shell of the host's machine is opened.

In an actual vulnerable program there might be more than one ways to open a shell to the host's machine. One program might be vulnerable to a few different techniques. These techniques are ret2libc, ret2syscall, and SROP. A graph of ROP can be simpler or more complex depending on the security measures a program has. If there is a security measure that completely blocks one specific technique, then the graph might become simpler due to the fact that there are no edges connecting the states for that technique to work. But a security measure might make the graph more complex if it does not completely block a technique, but instead only adds some extra steps and extra gadgets to make the technique work.

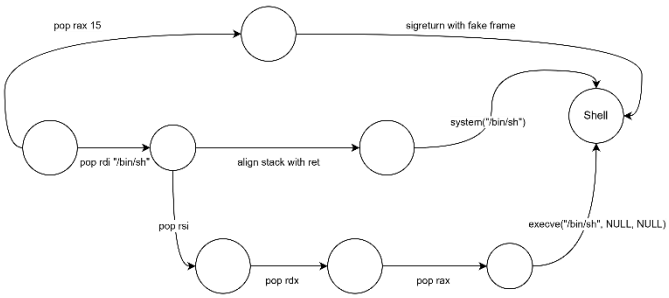


Fig 4.2 graph of an x86-64 linux program that is vulnerable to ret2libc, ret2syscall, and sigreturn

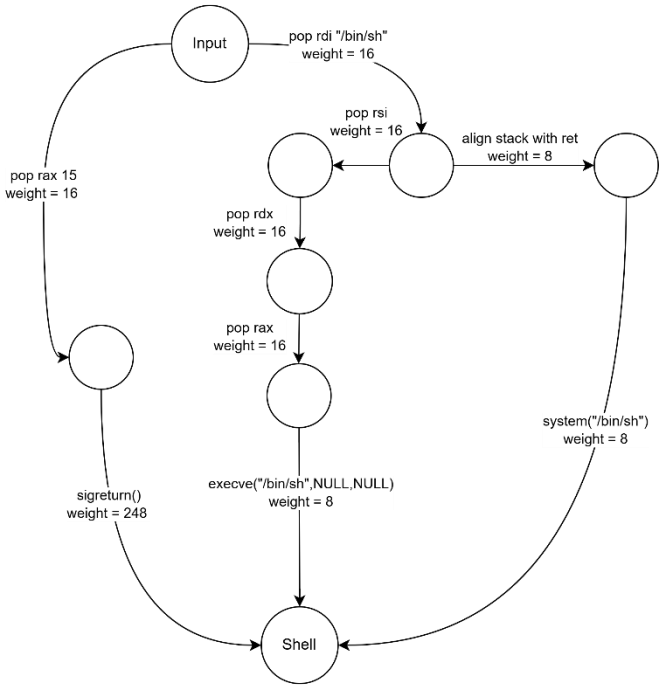
B. Weight assignment on edges

For a quantitative analysis to be possible, it is needed for a graph to be assigned weight on the edges. For a ROP graph there are multiple metrics that is usable for the assigned weight. One of those metrics that is the simplest is to weight the edge based on the amount of bytes the payload for a gadget need. To calculate the weight of an edge based on the payload size, we need to calcululate the size of the gadgets pointers itself, and the arguements that it will need. Using this metric, typically the weight of an edge would be multiple of 4 bytes or 8 bytes, depending on the system (32 bit or 64 bit). But there are also special conditions on which the size is not the multiple of 4 or 8 bytes. Although, in these special conditions a complication might arise, that which the stack becomes misaligned due to the non-typical size. To fix this an attacker would need to align the stack with the help of another gadget.

To calculate the weight of an edge, the formula below is

$$Weight = Return\ Pointer\ Size + Argument\ Size$$

with Return Pointer Size depending on the size of a pointer and Argument Size is the size of a pointer times the amount of arguments. With this formula we can add weight to our previous graph.



C. Defense Mitigations

An accurate graph representation is not only valuable for its descriptive power but also for its capacity to elucidate the impact of external forces. Within the context of our research, the true analytical strength of the state-space graph model is most profoundly demonstrated when used to formally describe and quantify the effects of modern defensive mitigations. This subchapter posits that security mechanisms such as Address Space Layout Randomization (ASLR) and Control-Flow Integrity (CFI) can be understood not as abstract concepts but as concrete mathematical transformations applied to the ROP graph. By modeling these defenses as specific graph operations,

we can move beyond qualitative assessments of their strength and toward a quantitative analysis of their efficacy in disrupting exploit construction.

1. The Impact of ASLR
   Address Space Layout Randomization (ASLR) is a foundational defense that aims to thwart memory corruption attacks by randomizing the base addresses of key memory segments, including the stack, heap, and shared libraries. Within our graph-theoretic framework, ASLR does not alter the fundamental topology of the ROP graph; the vertices (states) and the transitional logic of the edges (gadgets) remain intact within the binary's code. Instead, ASLR's effect manifests as a problem of partial observability. It effectively anonymizes the majority of the graph's edges by making their primary identifier—their memory address—a random variable.
2. Memory Leaks as Graph Discovery
   While ASLR obscures the graph, its protection is contingent on the secrecy of memory addresses, a property directly undermined by information disclosure vulnerabilities. An information leak is the mechanism by which an attacker pierces the veil of randomization, and its impact can be modeled as a process of graph discovery. When an attacker successfully leaks a single runtime address from a randomized memory region—for instance, the address of the puts function from a program's Global Offset Table—they have achieved far more than learning the location of one function.

D. Limitations of ROP in Graph Representation

While the state-space graph provides a powerful and elegant framework for formalizing Return-Oriented Programming, it warrants a critical evaluation of its inherent limitations and practical scalability. The utility of any theoretical model is bound by its underlying assumptions and its computational feasibility when applied to real-world problems. Where the abstraction fails to capture the full complexity of modern software execution—and by addressing the significant computational challenges that arise when attempting to apply this model to large-scale, contemporary binaries. Acknowledging these constraints is not a refutation of the model's value, but rather a necessary step in defining its scope and guiding its practical application.

1. Scalability
   The most significant barrier to the naive implementation of our model is the problem of state-space explosion. We have defined a vertex as a complete state of the machine's general-purpose registers. On a modern 64-bit architecture with sixteen or more registers, the theoretical number of unique states is (2^64)^16, a number so astronomically large as to be computationally indistinguishable from infinite. A graph with a vertex set of this magnitude is impossible to construct, store, or traverse. This reality dictates that any practical application of the model cannot operate on the complete, theoretical state-space.

   To render the problem tractable, the model must employ state abstraction. Rather than tracking the state of all registers, a goal-oriented search would only track the registers relevant to the desired outcome. For an execve system call, for instance, the state can be simplified to a tuple representing only the values of rax, rdi, rsi, and rdx, with all other registers treated as "don't care" variables. While this abstraction makes computation feasible, it is a crucial limitation. It simplifies the problem by ignoring the potential for side effects in other registers, which could disrupt the ROP chain in unforeseen ways. Therefore, while the theoretical model is exhaustive, its practical implementation must necessarily be a heuristic-driven approximation, trading completeness for computational feasibility.

2. Limitation on Dynamic Memory
   A second fundamental limitation arises from the model's static nature. The ROP graph is constructed through the static analysis of a binary as it exists on disk. This snapshot-in-time approach, however, fails to account for the dynamic realities of modern program execution. Many sophisticated software systems employ techniques that alter their own code at runtime, creating a significant gap between the analyzed binary and the executable code that an attacker actually targets in memory

3. Gadget Side Effects
   The simplicity that makes the graph model elegant also imposes limits on its fidelity in representing complex, low-level constraints. A prime example is the lackluster representation of a gadgets side effect. There are cases in which a gadget that is needed does not exist conveniently. Which is to say, that the gadget exists but it has some side effects that may affect the flow of the exploit. An example of a side effect would be a gadget that pops more registers than we need, which can potentially alter the behavior of the program in the direction the original exploit does not intend to.

4. Misalignment representation
   Many system calls and library functions, particularly on x86-64, require the stack pointer to be 16-byte aligned. Misalignment can lead to subtle or catastrophic failures in the ROP chain. This constraint is a global property of a path through the graph, dependent on the cumulative size of all preceding gadgets. It cannot be accurately represented as a simple, static weight on a single edge, as the need for alignment padding is context-dependent. A basic implementation of Dijkstra's algorithm on our weighted graph would fail to account for this, potentially generating "optimal" paths that are, in fact, non-functional.

V. GRAPH MODELLING IN CTFS

Capture the Flag (CTF) is a competition or exercise where participants find and exploit vulnerabilities in systems to capture "flags," which are pieces of information or strings of code. It's a hands-on way to develop and test cybersecurity skills,

simulating real-world attacks and defenses. CTFs can be competitive or used for educational purposes. and guiding its practical application.

In this chapter a real CTF challenge from the website picoctf.org will be represented using the graph model. The website will give us the binary program that is going to be exploited, as well as a netcat to connect to once an exploit script is developed. Specifically the challenge that we are going to exploit is challenge 179 from PicoGym.

The binary given does not have PIE enabled, but it does have ASLR enabled, which means that the GOT table itself is not randomized, but LIBC is randomized. Subsequently, the location of the string "/bin/sh" which is stored in LIBC is also randomized. In this case we will bypass ASLR using puts() to leak memory of the GOT entry of a function in libc. We will pass GOT entry of functions in LIBC that has already been called at least once. This is due to lazy binding, which means resolution of symbols (like function addresses) from shared libraries is deferred until the symbol is actually used for the first time during program execution. From disassembling the binary using GDB we can find that puts(), setresgid(), getegid(), and setbuf() all has been used before the program takes input.

Here, we can use ROP to leak two GOT entries of the functions that was mentioned. We can visualize all possible paths to calculate the base of LIBC using graph representation. The following figure will show exactly that.

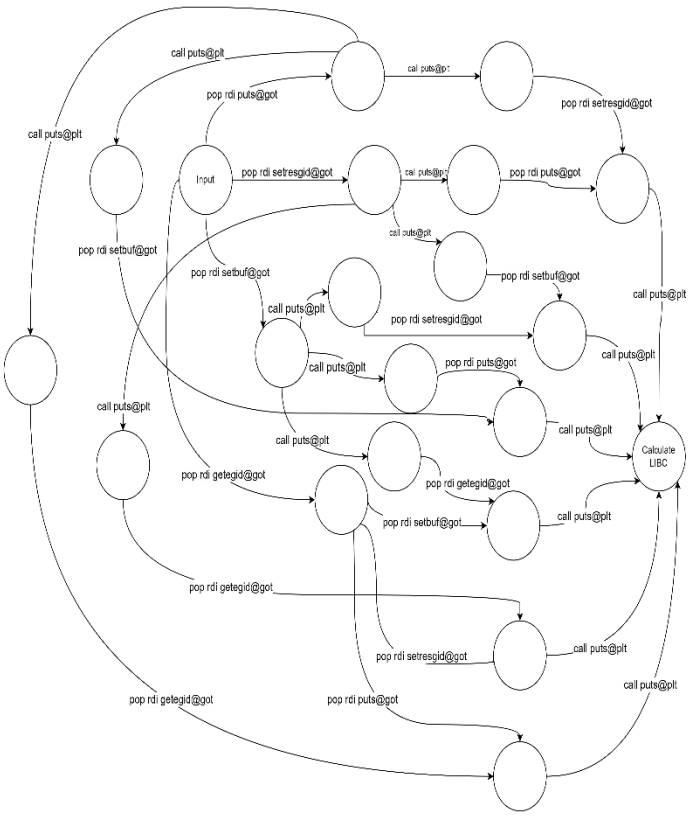


Fig 5.1 Graph of ROP chain to calculate LIBC

The reason the graph turns out to be cluttered is because it represents all the possible paths to calculate the base address of LIBC with all the available gadgets. An attacker can take any path and it will lead them to calculate the base of LIBC. This graph representation also have weights to each edge. Due to limited space for the graph, the weights are not written on the graph itself. Here, all edges that represent a "pop" gadget have weight of 16 bytes. And all call to puts@plt will have a weight of 8 bytes.

VI. Conclusion

Complex and often unintuitive process of Return-Oriented Programming can be formally modeled as a pathfinding problem within a state-space graph. By representing achievable machine states as vertices and gadgets as weighted, directed edges, we have transformed the art of exploit construction into a systematic and automatable science. This approach enables the direct application of established graph-theoretic algorithms, such as Dijkstra's, to not only determine the feasibility of an attack but also to automatically generate optimal payloads based on metrics like byte-efficiency. This model provides a unified, quantitative framework for analyzing the very nature of control-flow hijacking.

However powerful this approach is, its power lies in its role as a high-level abstraction. Its successful application requires acknowledging and mitigating the state-space explosion through goal-oriented search, understanding its inherent blindness to dynamic runtime code generation, and augmenting its pathfinding logic to handle complex, path-dependent constraints like stack alignment. These limitations do not invalidate the model; they define its character as a powerful but imperfect lens, providing a formal language to reason about a problem that remains, in its deepest complexities, a moving target.

VII. Appendix

Full PicoCTF writeup and exploit script: https://github.com/PTPB25/PTPB25-Archival/blob/main/Capture%20The%20Flag/Binary%20Exploitation/PicoCTF/Here's%20a%20Libc/Here's%20a%20LIBC.pdf

VIII. Acknowledgment

The author gives his upmost gratitude to Allah SWT for His endless guidance and mercy. Through His grace, the author has managed to complete this paper.

The author also gives his gratitude to Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecturer of Discrete Mathematics of class K02 IF1220, for his dedication and his guidance through his lectures throughout the first half of 2025.

References

[1] Ryan Roemer, "Return-Oriented Programming: Systems, Languages, and Applications" [Online]. Available: https://hovav.net/ucsd/dist/rop.pdf [Accessed: June 16, 2025].

[2] Rinaldi Munir, "Graf (Bagian 1)" [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf [Accessed: June 16, 2025].

[3] Huihoo, "sigcontext Struct Reference" [Online]. Available: https://docs.huihoo.com/doxygen/linux/kernel/3.7/structsigcontext.html [Accessed: June 17, 2025].

[4] Wenliang Du, *Computer Security: A Hands on Approach,* 3rd Edition.

[5] Jaehyuk Lee and Jinsoo Jang, “Hacking in Darkness: Return-oriented Programming against Secure Enclaves” [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk [Accessed: June 17, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Billie Bhaskara Wibawa 13524024