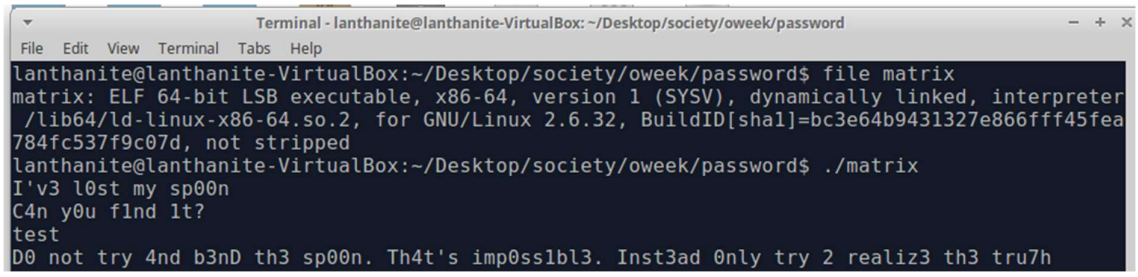
Matrix (150 pts)

Like any binary challenge, the first thing we do is verify our file type, and run it with test input.



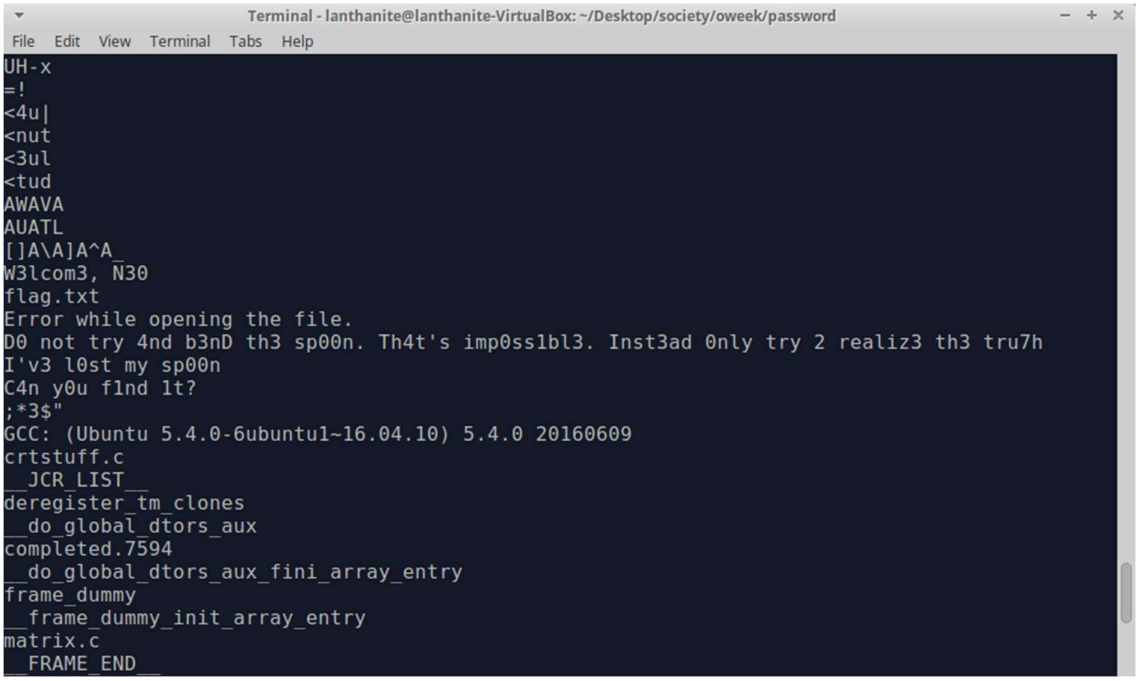
The important thing here's the first line – ELF 64-bit LSB executable, which tells us that this is a standard Linux binary. ELF files are roughly the Linux equivalent of Windows .exe files, or *executables.*

We see that the binary's asking for some input, and then giving us an error message. It looks like we have to guess a password of some sort to get the flag.

The important thing to note here is that the binary has to do some sort of verification – it has to compare our input to *something*, and then decide whether or not our input was correct.

That means the password should be inside the binary right?

We have a look using the *strings* utility to view human-readable strings inside the binary. If there's a password hidden in there, this should allow us to find it.



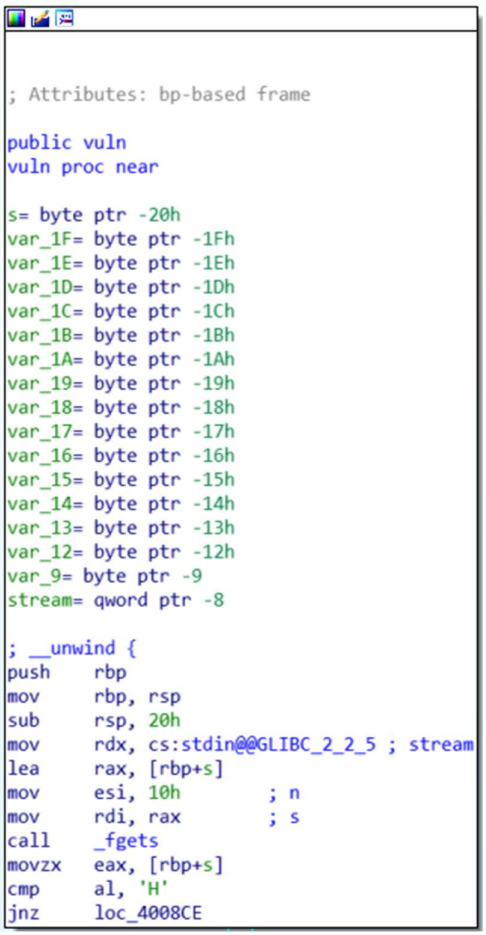
Looking at this, we don't see anything that looks like a password. "*Error while opening the file*" and *"flag.txt"* are probably related to reading the flag on a success.

But the password *has* to be in the binary somewhere. So what we do is open up the binary in IDA, a disassembly tool, to get a closer look at what precisely the binary's doing.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_20= qword ptr -20h
var_14= dword ptr -14h
rgid= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
call    _getegid
mov     [rbp+rgid], eax
mov     edx, [rbp+rgid] ; sgid
mov     ecx, [rbp+rgid]
mov     eax, [rbp+rgid]
mov     esi, ecx        ; egid
mov     edi, eax        ; rgid
call    _setresgid
mov     edi, offset aIV3L0stMySp00n ; "I'v3 l0st my sp00n"
call    _puts
mov     edi, offset aC4nY0uF1nd1t ; "C4n y0u f1nd 1t?"
call    _puts
mov     eax, 0
call    vuln
mov     eax, 0
leave
retn
; } // starts at 4008DB
main endp
```

If that looks like gibberish to you, don't worry. It's the x86 assembly instructions that the binary is composed of – the lowest-level human-readable code. Notice anything?

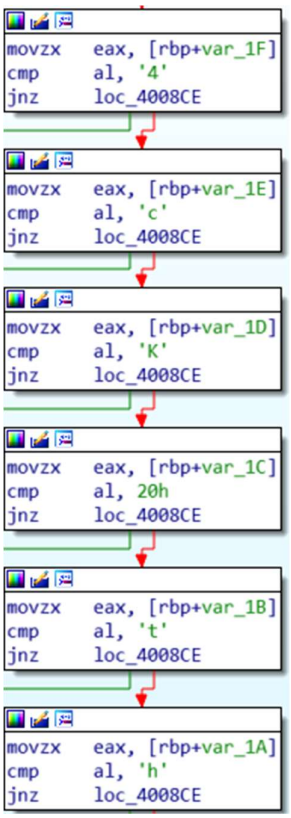
That *call vuln* line looks suspicious. Let's take a further look.

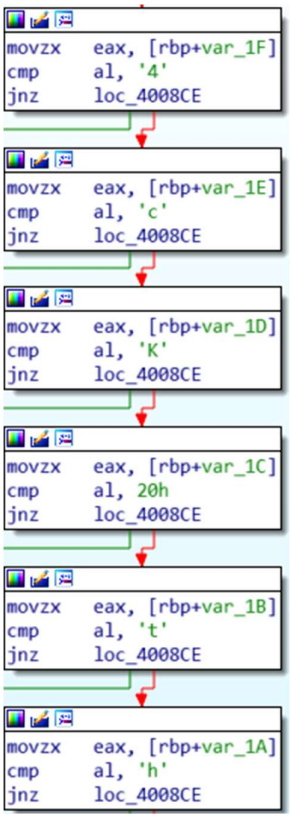


```
; Attributes: bp-based frame

public vuln
vuln proc near

s= byte ptr -20h
var_1F= byte ptr -1Fh
var_1E= byte ptr -1Eh
var_1D= byte ptr -1Dh
var_1C= byte ptr -1Ch
var_1B= byte ptr -1Bh
var_1A= byte ptr -1Ah
var_19= byte ptr -19h
var_18= byte ptr -18h
var_17= byte ptr -17h
var_16= byte ptr -16h
var_15= byte ptr -15h
var_14= byte ptr -14h
var_13= byte ptr -13h
var_12= byte ptr -12h
var_9= byte ptr -9
stream= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     rdx, cs:stdin@@GLIBC_2_2_5 ; stream
lea     rax, [rbp+s]
mov     esi, 10h        ; n
mov     rdi, rax        ; s
call    _fgets
movzx   eax, [rbp+s]
cmp     al, 'H'
jnz     loc_4008CE
```

The green lines starting with *var_X* signify local variables. That's a *lot* of local variables.

And what's that at the bottom? A "compare" instruction with the letter 'H'? Let's take a closer look.


movzx eax, [rbp+var_1F]
cmp al, '4'
jnz loc_4008CE
movzx eax, [rbp+var_1E]
cmp al, 'c'
jnz loc_4008CE
movzx eax, [rbp+var_1D]
cmp al, 'K'
jnz loc_4008CE
movzx eax, [rbp+var_1C]
cmp al, 20h
jnz loc_4008CE
movzx eax, [rbp+var_1B]
cmp al, 't'
jnz loc_4008CE
movzx eax, [rbp+var_1A]
cmp al, 'h'
jnz loc_4008CE


movzx eax, [rbp+var_1F]
cmp al, '4'
jnz loc_4008CE
movzx eax, [rbp+var_1E]
cmp al, 'c'
jnz loc_4008CE
movzx eax, [rbp+var_1D]
cmp al, 'K'
jnz loc_4008CE
movzx eax, [rbp+var_1C]
cmp al, 20h
jnz loc_4008CE
movzx eax, [rbp+var_1B]
cmp al, 't'
jnz loc_4008CE
movzx eax, [rbp+var_1A]
cmp al, 'h'
jnz loc_4008CE

Yikes.

What we see above is a series of chained *if/else* instructions, with each comparing one of the local variables to a letter, and only continuing if they match. Sounds awfully like a password check – and if we read through them, we get “H4cK th…” which looks awfully like a password.

The password wouldn’t have shown up when we ran *strings* because the password doesn’t exist as a whole string – just a series of individual letters.

Following the chain to the end, we get our password and, entering it, get our flag.

```
lanthanite@lanthanite-VirtualBox:~/Desktop/society/oweek/password$ ./matrix
I'v3 l0st my sp00n
C4n y0u f1nd 1t?
H4cK th3 Pl4n3t
W3lcom3, N30
OWEEK{It_is_n0T_tH3_sp00n_tH4t_b3nds,_1t_is_0nly_y0urs3lf}
```

Was the above challenge just a really long if statement? Yes. Yes it was.

```
void vuln(){

    char buffer[17];
    fgets(buffer, 16, stdin);
    if(buffer[0] == 'H' && buffer[1] == '4' && buffer[2] == 'c' && buffer[3] == 'K' && buffer[4] == ' '
    && buffer[5] == 't' && buffer[6] == 'h' && buffer[7] == '3' && buffer[8] == ' ' && buffer[9] == 'P'
    && buffer[10] == 'l' && buffer[11] == '4' && buffer[12] == 'n' && buffer[13] == '3' && buffer[14] == 't') {
        printf("W3lcom3, N30\n");
        FILE *fp;
        fp = fopen("flag.txt", "r");

        if (fp == NULL){
            perror("Error while opening the file.\n");
            exit(EXIT_FAILURE);
        }
        char ch;
        while((ch = fgetc(fp)) != EOF) printf("%c", ch);
        printf("\n");
    }
    else {
        printf("D0 not try 4nd b3nD th3 sp00n. Th4t's imp0ss1bl3. Inst3ad 0nly try 2 realiz3 th3 tru7h\n");
    }
}
```