

Very Hard 99% you can't do it

Challenge Author: Bobx

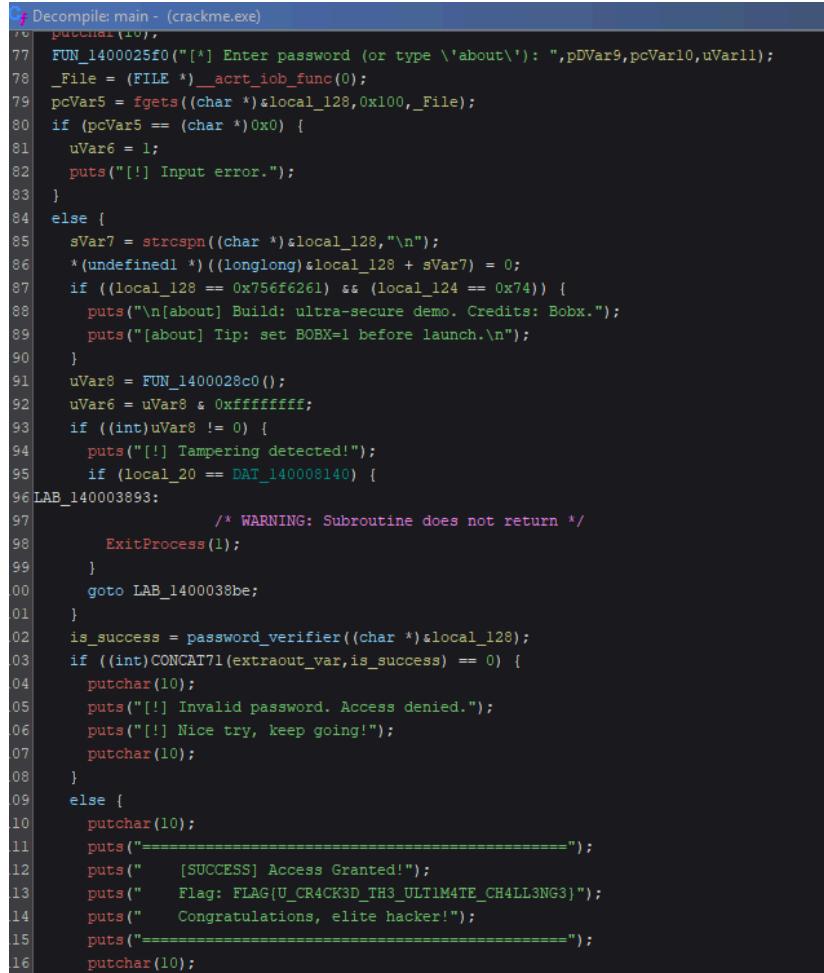
Writeup by scaredandalone

If you do your rudimentary checks like *strings* on the executable, you will find that the flag for the challenge is plain text.

However, that is no fun so let's instead reverse the password checker function.

Looking through the entry function and following function calls, we can find the “main” function which contains all of the logic.

Figure 1: Main re-compiled view



The screenshot shows the decompiled assembly code for the main function of the crackme.exe executable. The code is written in C-like pseudocode. It starts with a call to `putchar(10)`, followed by a call to `FUN_1400025f0` which prompts for a password. The password is read from the file `_File` using `fgets`. If the password is empty (0x0), an error message is printed. Otherwise, the password is checked against a hardcoded value at `local_128`. If they match, the program prints a success message and the flag. If they don't match, it prints an invalid password message. The code ends with a call to `ExitProcess`.

```
0: 00401000 putchar(10);
1: 00401005 FUN_1400025f0("[*] Enter password (or type '\about\'): ",pDVar9,pcVar10,uVar11);
2: 00401010 _File = (FILE *)__acrt_iob_func(0);
3: 00401015 pcVar5 = fgets((char *)slocal_128,0x100,_File);
4: 00401020 if (pcVar5 == (char *)0x0) {
5: 00401025     uVar6 = 1;
6: 00401030     puts("[!] Input error.");
7: 00401035 }
8: 00401040 else {
9: 00401045     sVar7 = strcspn((char *)slocal_128,"\\n");
10: 00401050     *(undefined *)((long long)slocal_128 + sVar7) = 0;
11: 00401055     if ((local_128 == 0x756f6261) && (local_124 == 0x74)) {
12: 00401060         puts("\n[about] Build: ultra-secure demo. Credits: Bobx.");
13: 00401065         puts("[about] Tip: set BOBX=1 before launch.\n");
14: 00401070     }
15: 00401075     uVar8 = FUN_1400028c0();
16: 00401080     uVar6 = uVar8 & 0xffffffff;
17: 00401085     if ((int)uVar8 != 0) {
18: 00401090         puts("[!] Tampering detected!");
19: 00401095         if (local_20 == DAT_140008140) {
20: 00401100             LAB_140003893:
21: 00401105                 /* WARNING: Subroutine does not return */
22: 00401110                 ExitProcess(1);
23: 00401115             }
24: 00401120             goto LAB_1400038be;
25: 00401125         }
26: 00401130         is_success = password_verifier((char *)slocal_128);
27: 00401135         if ((int)CONCAT71(extraout_var,is_success) == 0) {
28: 00401140             putchar(10);
29: 00401145             puts("[!] Invalid password. Access denied.");
30: 00401150             puts("[!] Nice try, keep going!");
31: 00401155             putchar(10);
32: 00401160         }
33: 00401165     else {
34: 00401170         putchar(10);
35: 00401175         puts("=====================");
36: 00401180         puts("      [SUCCESS] Access Granted!");
37: 00401185         puts("      Flag: FLAG(U_CR4CK3D_TH3_ULT1M4TE_CH4LL3NG3)");
38: 00401190         puts("      Congratulations, elite hacker!");
39: 00401195         puts("=====================");
40: 00401200         putchar(10);
```

We can see the flag that I was talking about earlier and the password verification function that leads to it upon success.

Figure 2: Build password function

```
Decompile: build_password - (crackme.exe)
.3 undefined local_40;
.4 undefined8 uStack_40;
.5 undefined8 uStack_38;
.6 undefined8 uStack_30;
.7 longlong local_20;
.8 byte seed_value;
.9
.10 local_20 = DAT_140008140;
.11 local_68 = 0x8783445371865073;
.12 uStack_60 = 0x484c8677454d4c85;
.13 uStack_58 = 0x8280825f4c745086;
.14 uStack_50 = 0x218186;
.15 local_48 = 0;
.16 uStack_40 = 0;
.17 uStack_38 = 0;
.18 uStack_30 = 0;
.19 proc_ID = GetCurrentProcessId();
.20 tickCount = GetTickCount();
.21 seed_value = (byte)proc_ID ^ (byte)tickCount;
.22 bVar1 = 0x73;
.23 pbVar2 = (byte *)&local_68;
.24 do {
.25     *pbVar2 = bVar1 ^ seed_value;
.26     bVar1 = pbVar2[1];
.27     pbVar2 = pbVar2 + 1;
.28 } while (bVar1 != 0);
.29 pbVar2 = (byte *)&local_68;
.30 bVar1 = (byte)local_68;
.31 while (bVar1 != 0) {
.32     *pbVar2 = bVar1 ^ seed_value;
.33     bVar1 = pbVar2[1];
.34     pbVar2 = pbVar2 + 1;
.35 }
```

We can see something very odd going on here.

The function is trying to hide a hard coded password by using some “dynamic” logic.

First the four 8-byte constants are placed into a local buffer (local_68, uStack_60, uStack_58, uStack_50), which together form the raw encoded bytes of the secret.

Then it reads the current process ID and the tick count since (system) startup, XORs their lower byte to get a single “seed” byte (seed_value).

This key is then used as a key in two passes over the buffer.

In the first loop, it walks forward through the bytes starting at local_68, repeatedly setting the byte at *pbVar2 = bVar1 ^ seed_value, where bVar1 is initially 0x73 and then updated to the next byte each time.

This continues until it reaches a null terminator.

However, immediately after the first pass, the second loop resets the pointer to the start of the buffer and does another walk.

Because XOR is its own inverse, and because of how bVar1 is updated from the transformed data, the second pass effectively undoes the seed-dependent scrambling from the first pass.

As a result, the final contents of the buffer converge to a fixed byte sequence that no longer depends on the process ID or tick count.

The double-XOR logic is constructed so that the final string copied out of local_68, is actually constant across all environments. Meaning, that the PID and tickcount do not actually contribute to the generation.

Password (Raw hex bytes):

73 50 86 71 53 44 83 87 85 4C 4D 45 77 86 4C 48 86 50 74 4C 5F 82 80 82 86 81 21

Flag:

FLAG{U_CR4CK3D_TH3_ULT1M4TE_CH4LL3NG3}