

# ITFest 2020 CTF writeup

Bodea Coralia, Vîjdea Cristian

## PROBLEM 1

---

### MISSION 1

binary1.dat looks like base64. We try to decode it and get the attached message.

```
C:\Users\cvijdea.TSR\Downloads\Problem1>base64 -d binary1.dat
Welcome to IT*FEST*2020!

We're excited to have you here - and we're sure you will enjoy the challenges!

~Stay safe & enjoy the fun!
Agent 000 @ IT Fest 2020

P.S.: There's no codes in here. But... they're hidden somewhere in here!

Motto: "When the salt is sour, the whole world sweetens up!"
```

Flag: **IT\*FEST\*2020**

## MISSION 2

This doesn't work for the other 2 files... keep looking :(

Running `strings` on the binary we see there are some sections named UPX.

```
C:\Users\cvijdea.TSR\Downloads\Problem1>strings CTF1.exe | head
!This program cannot be run in DOS mode.
UPX0
UPX1
UPX2
3.96
UPX!
dHjbdH
dZRJ
B:2H
dH*"dH
```

Therefore we proceed with unpacking the executable using upx (<https://upx.github.io/>):

```
C:\Users\cvijdea.TSR\Downloads\Problem1>../upx/upx-3.96-win64/upx.exe -d CTF1.exe
          Ultimate Packer for eXecutables
          Copyright (C) 1996 - 2020
UPX 3.96w      Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

  File size      Ratio      Format      Name
  -----
  245248 <-      81408      33.19%     win64/pe     CTF1.exe

Unpacked 1 file.
```

After unpacking it turns out that the executable is a huge Pascal program that would be too hard to reverse engineer using IDA. Let's try something else...

From the statement of Mission 2 we can guess that `binary2.dat` is generated using an invocation like `ctf1 -c original_data.bin`. So let's try and figure out what that does.

First of all, we try to encode a simple alphabet string. This turns into another string that seems base64 encoded, which decodes to something that looks suspiciously like a substitution cipher.

```
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ cat teest2.txt
abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ ctf1 -c teest2.txt
OT07PD0+PzAxMjMONTY3KckqKywtLi8gISJoawprYwxtb2BhOT07PD0+PzAxMjMONTY3KckqKywtLi8gISJoawprYwxtb2Bh
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ ctf1 -c teest2.txt | base64 -d
9:;<=>?01234567()*+,-./ !"hijklmo`a9:;<=>?01234567()*+,-./ !"hijklmo`a
```

We can then write a simple Python program to generate a file containing the set of printable ASCII characters and encode it using the program.

```
alphabet = bytes(range(32, 128)) + b'\n\r'
Path('alphabet.bin').write_bytes(alphabet)
alphabet_encoded = base64.b64decode(check_output('ctf1.exe -c alphabet.bin'))
```

By reversing this mapping we can get the reverse substitution cipher.

```
reverse_cipher = {}
for idx, encoded in enumerate(alphabet_encoded):
    if idx < len(alphabet):
        reverse_cipher[encoded] = alphabet[idx]

ciphertext = base64.b64decode(Path(sys.argv[1]).read_bytes())

decoded = bytearray()
for e in ciphertext:
    decoded.append(reverse_cipher[e])

print(decoded.decode('utf-8'))
```

The complete script is attached as `subst.py`

By running it we get the following output:

```
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ python subst.py binary2.dat
Congratulations for making it this far, Agent Johnny English!

This is very impressive. We're sure you're on the right track.
Don't worry, the codes are ALMOST at your fingertips.
~The guys at IT*FEST :-)
```

## MISSION 3

Using the same principles as in the previous missions, we generate some inputs which we give to `ctf1 -x`. The output of this command appears again to be base64 encoded.

```
a -> EmJ/ -> [18, 98, 127]
ab -> Eg14 -> [18, 13, 120]
abc -> Eg0w -> [18, 13, 22]
abcd -> Eg0wFn5l -> [18, 13, 22, 22, 126, 101]
abcde -> Eg0wFhZi -> [18, 13, 22, 22, 22, 98]
abcdef -> Eg0wFhYJ -> [18, 13, 22, 22, 22, 9]
abcdefg -> Eg0wFhYJEn95 -> [18, 13, 22, 22, 22, 22, 9, 18, 127, 121]
abcdefgh -> Eg0wFhYJEhp+ -> [18, 13, 22, 22, 22, 22, 9, 18, 26, 126]
```

When looking at the outputs we can see that they remain stable when adding additional characters. We can also note that the output is always padded so that its length is a multiple of 3, likely as an artifact of base64 encoding.

However when generating repeating inputs we see something interesting:

```
a -> EmJ/ -> [18, 98, 127]
aa -> Eg54 -> [18, 14, 120]
aaa -> Eg4U -> [18, 14, 20]
aaaa -> Eg4UE35l -> [18, 14, 20, 19, 126, 101]
aaaaa -> Eg4UEXji -> [18, 14, 20, 19, 18, 98]
aaaaaa -> Eg4UEXIO -> [18, 14, 20, 19, 18, 14]
aaaaaaa -> Eg4UEXIOFH95 -> [18, 14, 20, 19, 18, 14, 20, 127, 121]
aaaaaaaa -> Eg4UEXIOFBN+ -> [18, 14, 20, 19, 18, 14, 20, 19, 126]
```

It appears that the algorithm here is no longer a simple substitution cipher, but is also somehow related to the position of the character in the string. We can also see a repeating pattern every 4 characters - the encoding for "aaaaaaa" is equal to 2x "aaaa".

We can then form a hypothesis that this is just the same cipher as in Mission 2, but using 4 different substitutions depending on the position in the string. We can then use another Python script to generate 4 reverse ciphers, for each printable character.

The decoding function must then simply change its reverse cipher according to `idx % 4`:

```
def decode(ciphertext):
    result = bytearray()
    for idx, e in enumerate(ciphertext):
        result.append(reverse_ciphers[idx % 4].get(e, 32))

    return result.decode()
```

The complete script is attached as `x.py`.

By running the script we get the following output:

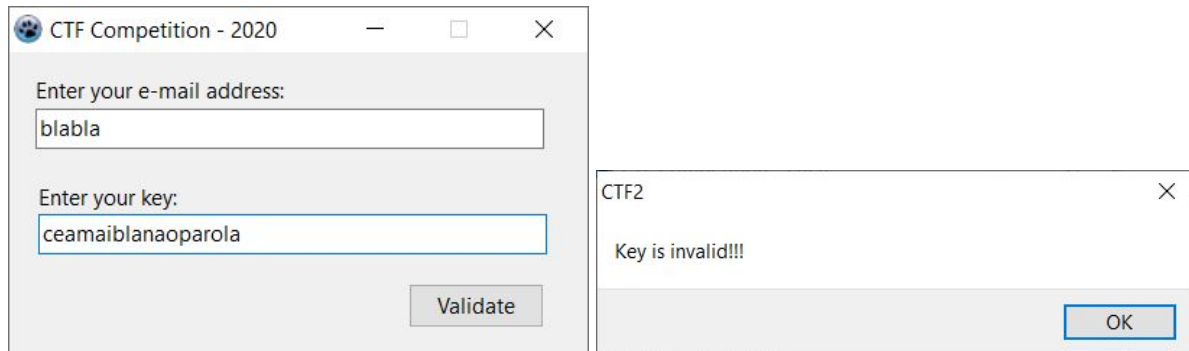
```
cviijdea@CVIJDEA-L2 C:\Projects\itfest-ctf  
$ python x.py binary3.dat  
Amazing work!!! Your nuclear codes are: WAR*@*IT*FEST*2020
```

**Final flag:** WAR\*@\*IT\*FEST\*2020

## PROBLEM 2

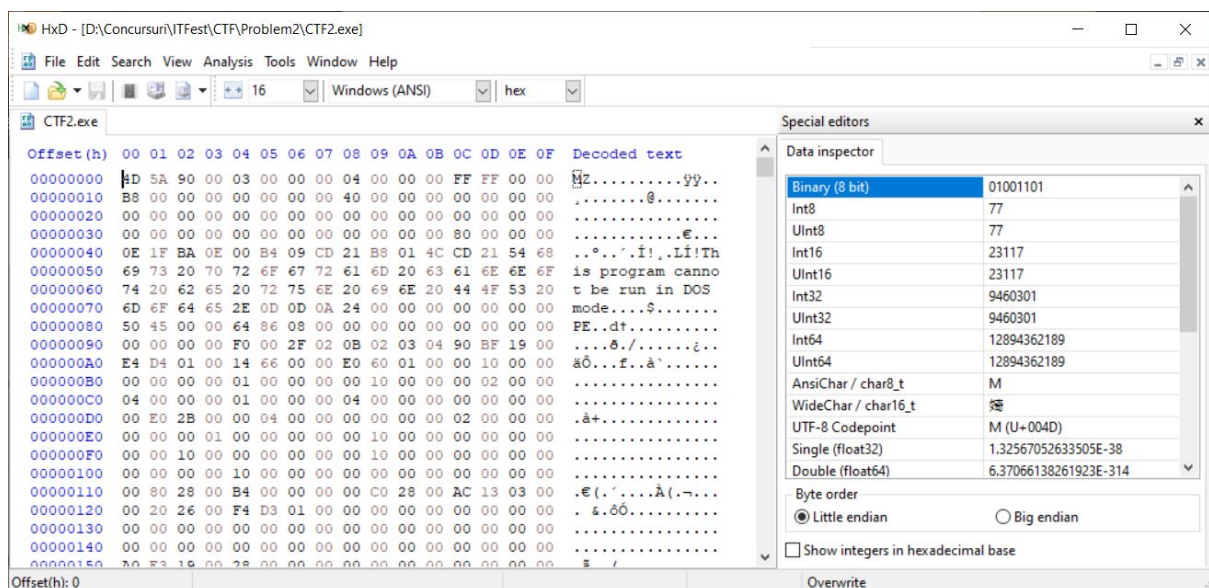
### MISSION 1

First we run `CTF2.exe` and give random strings as inputs for both email and password in order to see what happens.



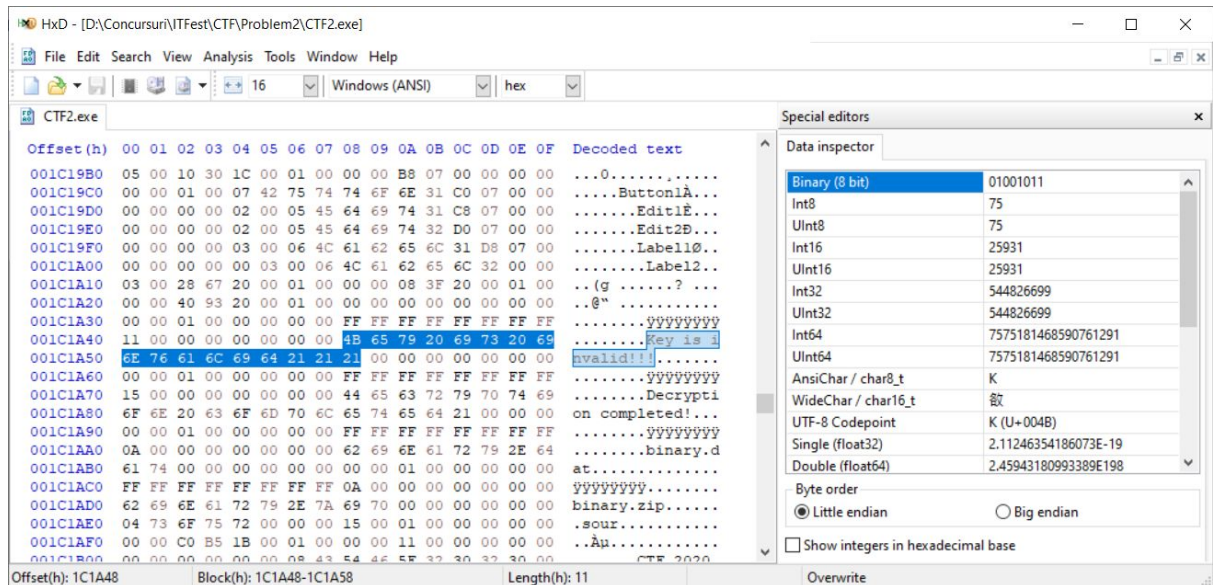
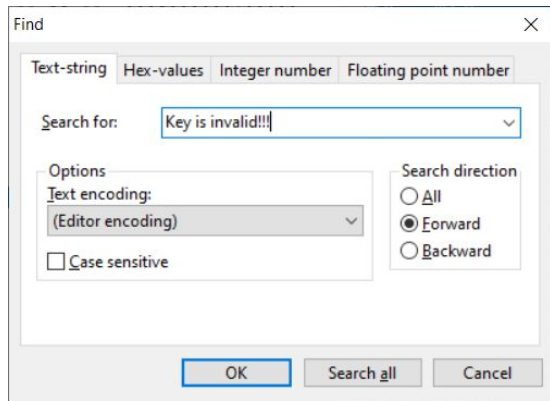
We notice that the string *"Key is invalid!!!"* is the string that needs to be converted to *"IT\*FEST rulz!"*.

Then we open `CTF2.exe` using the program **HxD** which is a hex editor.

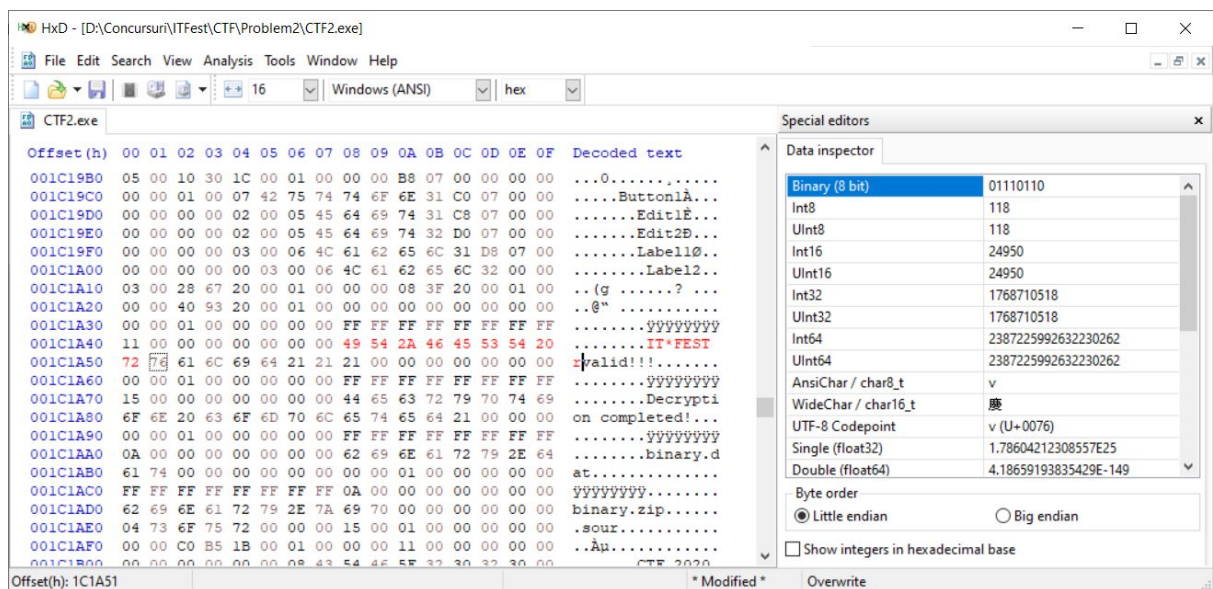


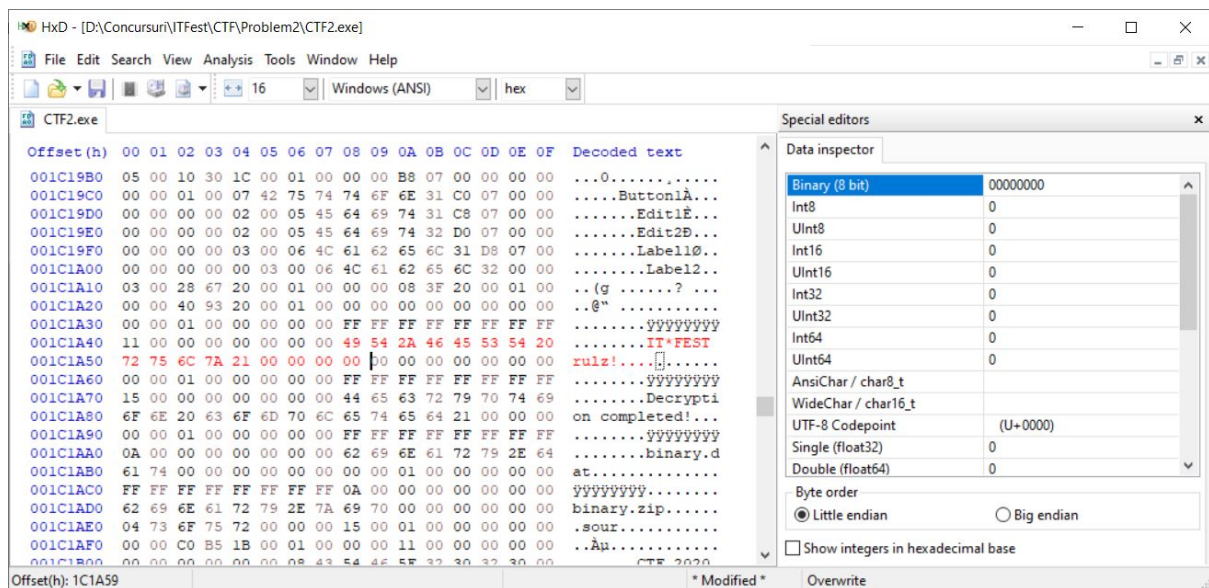
Now that we know the message we search for the string *"Key is invalid!!!"*.



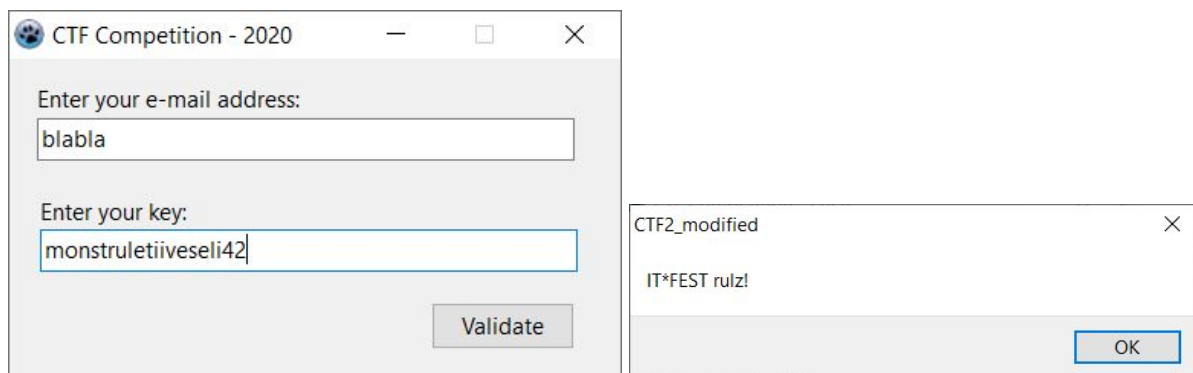


We replace "Key is invalid!!!" with "IT\*FEST rulz!" and the remaining characters (because "Key is invalid!!!" has 17 characters and "IT\*FEST rulz!" has only 13 characters) with the hex value 0x00 and we save the file (as CTF2\_mission1.exe).





We run CTF2\_mission1.exe and give random strings as inputs for both email and password to see if we got it right.



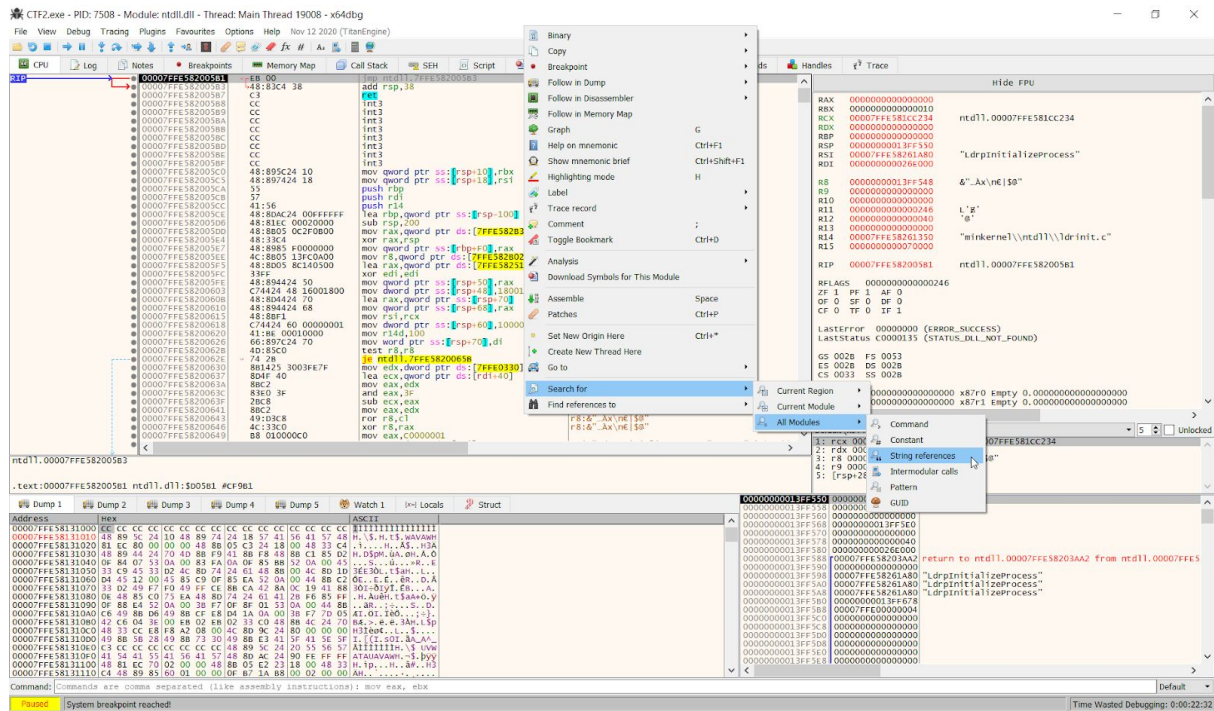
Yay! It worked!

**Solution:** CTF2\_mission1.exe (you can find our modified file in the same folder as this writeup)

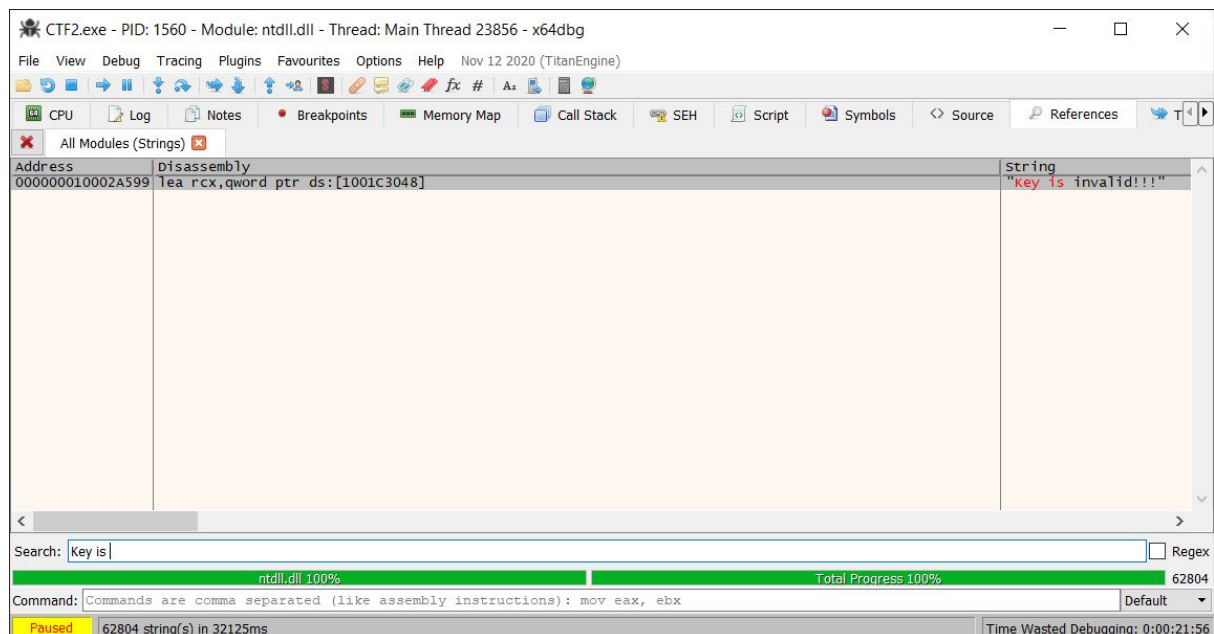


## MISSION 2

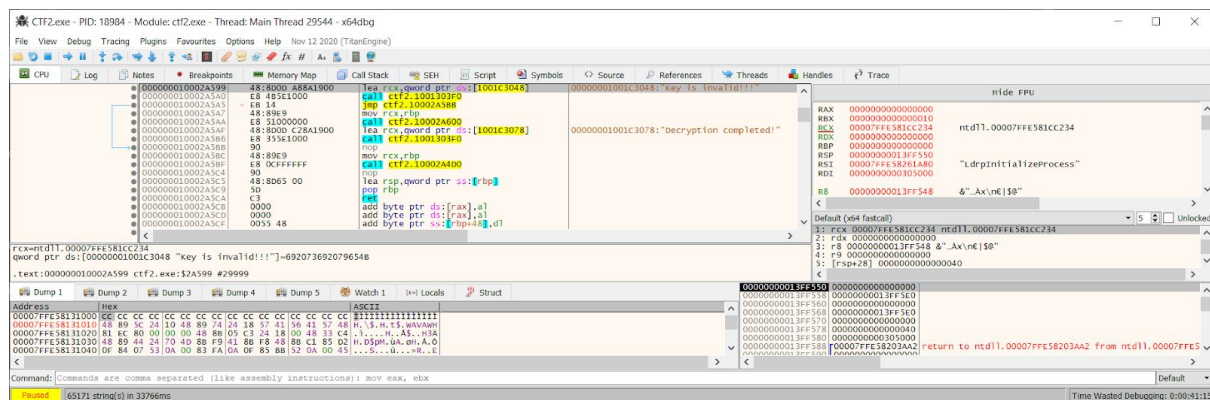
We open CTF2.exe using the program x64dbg which is an open-source x64/x32 debugger for windows.



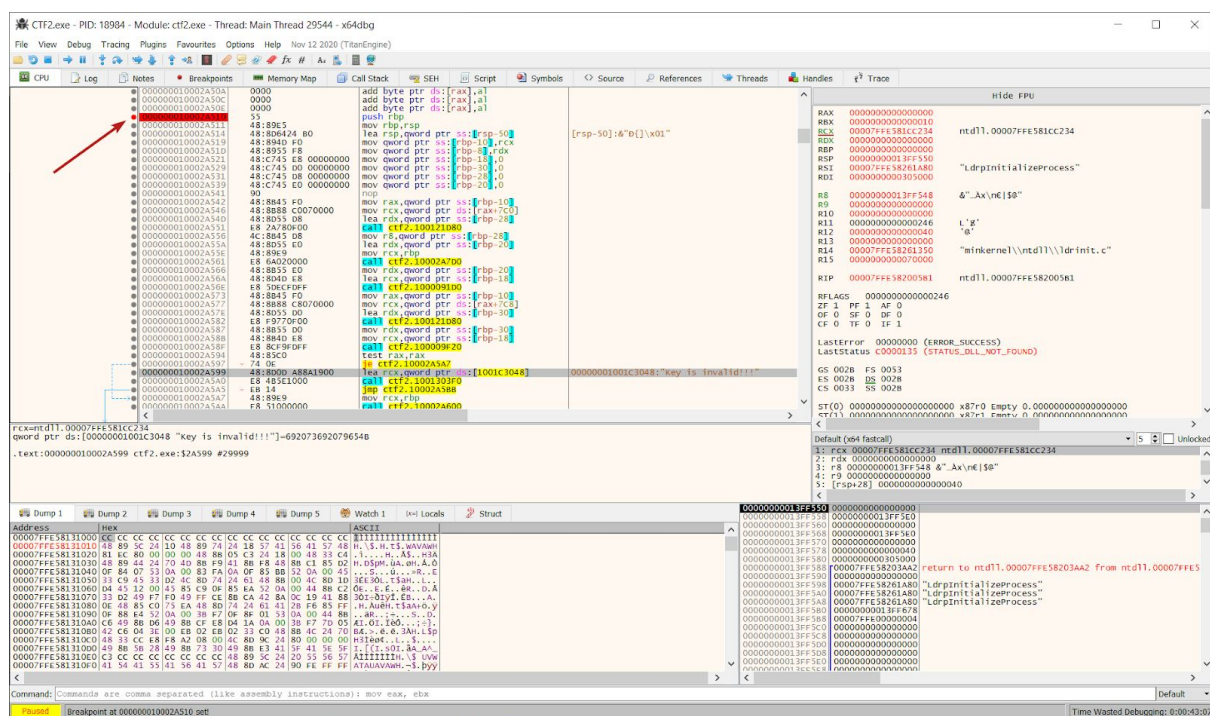
We search for string references, we get the list with all the strings contained by the executable and we search for "Key is invalid!!!".



We double click our search result to see where it takes us.



We set a breakpoint at the beginning of the function which checks our password.



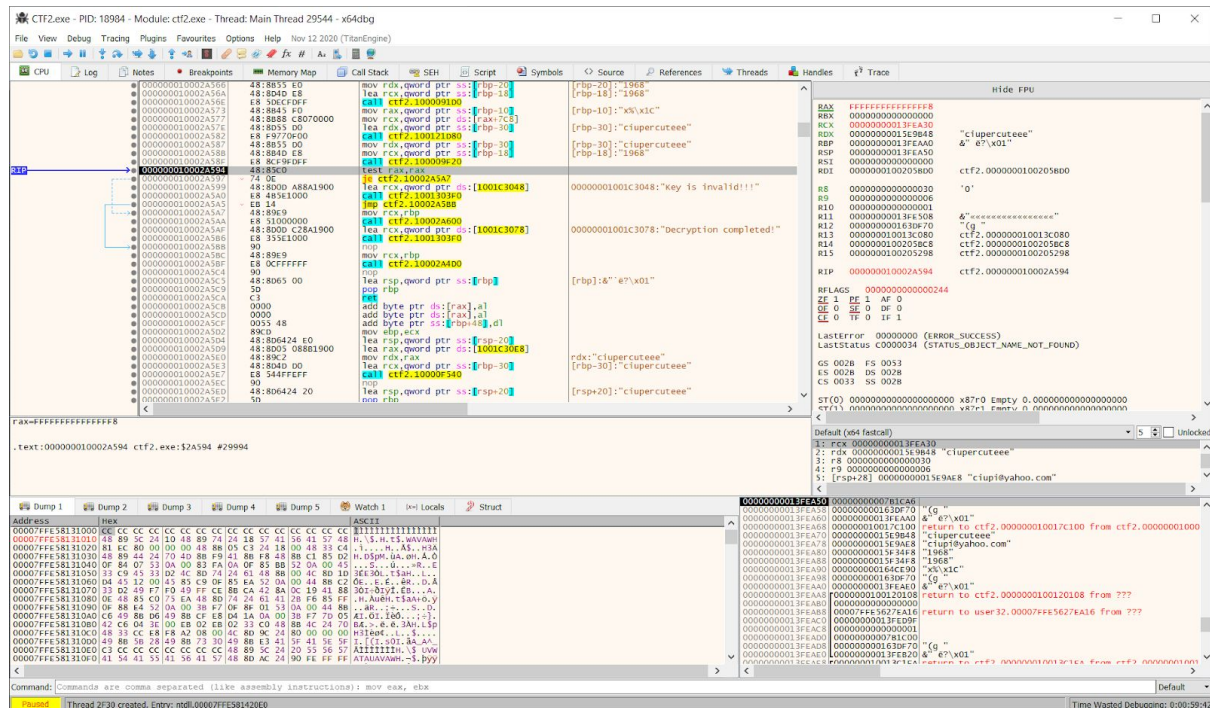
We press F9 to run the program and we give random strings as inputs for both email and password.

Enter your e-mail address:

Enter your key:



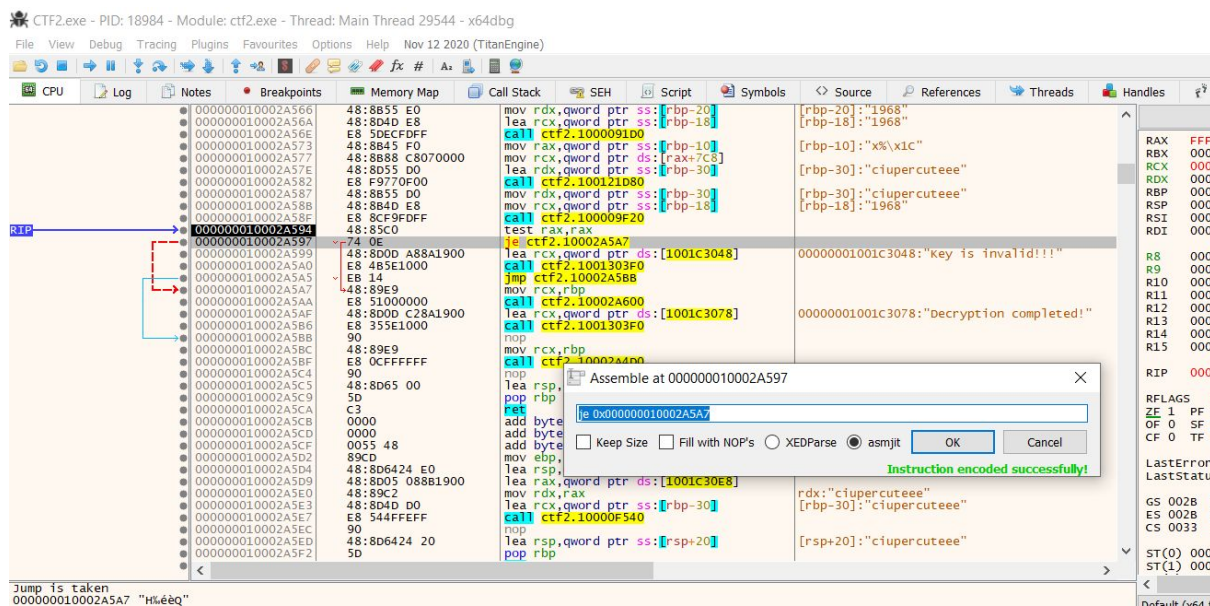
We reach our breakpoint and start running the program step by step using Step into (F7) and Step over (F8) in order to understand what it does.

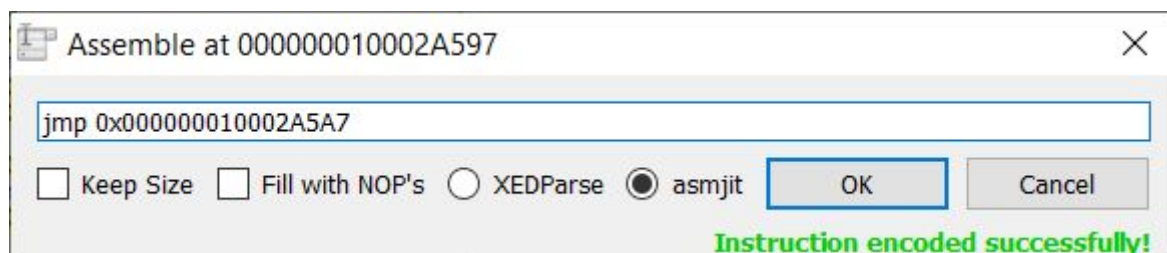


As we can see, if the two values that are being compared are equal, we get the message "Decryption completed!", otherwise we get the message "Key is invalid!!!".

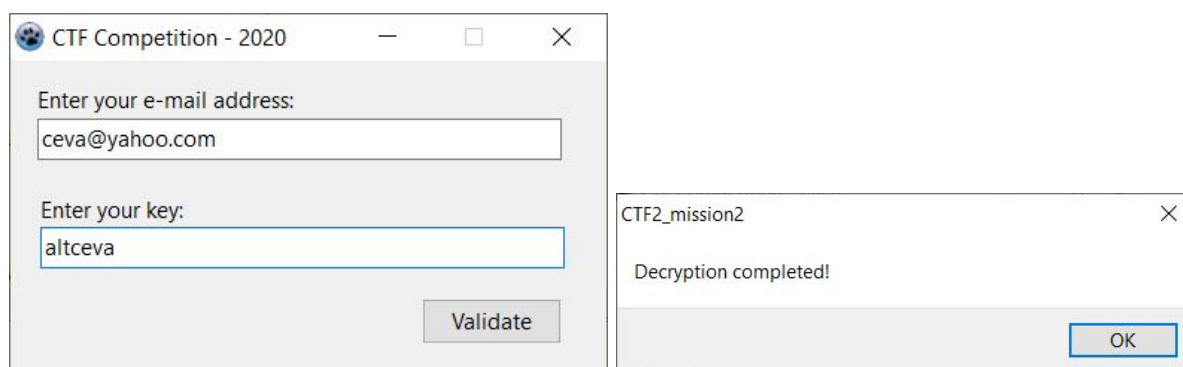
Using the built in assembler, we can change the jump instruction. That means that if we change it to `jmp` instead of `je`, it will always show that the password is valid regardless of what we enter.

We press the spacebar to change the instruction.





After we click OK, the instruction is immediately modified in memory. We export the patched file and we test it to make sure it does the job.



The message we get after clicking Validate is "Decryption completed!". We also notice a zip file named `binary.zip` appears in our folder, which is a password-protected ZIP archive containing `binary.txt` inside it.

**Solution:** `CTF2_mission2.exe`

## MISSION 3

By analysing the program during runtime we can see that the actual algorithm for validating the key is simply comparing the sum of the ASCII codes of the characters in the username, with a prepended salt (hardcoded as "sour"), against the key. So, for example, the username "testmail" corresponds to the password "1324" = 115 + 111 + 117 + 114 + 116 + 101 + 115 + 116 + 109 + 97 + 105 + 108

```

0000000010002A7D9 48 89 4D E8 mov qword ptr ss:[rbp-18],rcx
0000000010002A7DD 48 89 55 F0 mov qword ptr ss:[rbp-10],rdx
0000000010002A7E1 4C 89 45 F8 mov qword ptr ss:[rbp-8],r8
0000000010002A7E5 48 88 4D F8 mov rcx,qword ptr ss:[rbp-8]
0000000010002A7E9 E8 82 E9 FD FF call ctf2.1000091A0
0000000010002A7EE 90 nop
0000000010002A7EF C7 45 D8 00 00 00 00 dword ptr ss:[rbp-28],0
0000000010002A7F6 C7 45 E0 01 00 00 00 dword ptr ss:[rbp-20],1
0000000010002A7FD 83 6D E0 01 sub dword ptr ss:[rbp-20],1
0000000010002A801 66 66 66 90 nop
0000000010002A805 7C 5E 30 90 nop
0000000010002A808 83 45 E0 01 add dword ptr ss:[rbp-20],1
0000000010002A80C 4F 86 55 E0 movzx edx,byte ptr ss:[rbp-20]
0000000010002A810 48 8D 05 C9 88 19 00 lea rax,qword ptr ds:[1001C30E0]
0000000010002A817 0F 86 04 10 movzx eax,byte ptr ds:[rax+rdx]
0000000010002A81B 01 45 D8 add dword ptr ss:[rbp-28],eax
0000000010002A81E 83 7D E0 04 cmp dword ptr ss:[rbp-20],4
0000000010002A822 7C E4 jn ctf2.10002A808
0000000010002A828 48 85 C0 test rax,rax
0000000010002A82B 74 04 je ctf2.10002A831
0000000010002A82D 48 8B 40 F8 mov rax,qword ptr ds:[rax-8]
0000000010002A831 C7 45 E0 01 00 00 00 mov dword ptr ss:[rbp-20],1
0000000010002A835 3B 45 E0 cmp eax,dword ptr ss:[rbp-20]
0000000010002A838 7C 24 jn ctf2.10002A861
0000000010002A83D 83 6D E0 01 sub dword ptr ss:[rbp-20],1
0000000010002A841 66 66 66 90 nop
0000000010002A845 66 66 90 nop
0000000010002A848 83 45 E0 01 add dword ptr ss:[rbp-20],1
0000000010002A84C 4F 86 55 F8 movzx edx,byte ptr ss:[rbp-8]
0000000010002A850 0F 86 54 0A FF movsxd rcx,dword ptr ss:[rbp-20]
0000000010002A854 08 54 0A FF movzx edx,byte ptr ds:[rdx+rcx-1]
0000000010002A859 01 55 D8 add dword ptr ss:[rbp-28],edx
0000000010002A85C 3B 45 E0 cmp eax,dword ptr ss:[rbp-20]
0000000010002A85F 7C E4 jn ctf2.10002A848
0000000010002A861 48 8B 4D F0 mov rcx,qword ptr ss:[rbp-10]
0000000010002A865 8B 55 D8 mov edx,dword ptr ss:[rbp-28]
0000000010002A868 E8 C3 2A 02 00 call ctf2.10004D330
0000000010002A86B 90 nop
0000000010002A86E 48 89 E9 mov rcx,rbp
0000000010002A871 E8 3A FF FF FF call ctf2.10002A7B0
0000000010002A876 90 nop
0000000010002A877 48 8D 65 00 lea rsp,qword ptr ss:[rbp]
0000000010002A87B 5D pop rbp
0000000010002A87C C3 ret
0000000010002A87F 00 00

```

Further analysis of the decryption function reveals that the binary.zip file is produced by XOR-ing the binary.dat file 2 bytes at a time with a hardcoded value of 0xC54E.

```

0000000010002A69C C7 45 F0 4E 00 00 00 mov dword ptr ss:[rbp-10],4E
0000000010002A6A3 C7 45 E8 C5 00 00 00 mov dword ptr ss:[rbp-18],C5
0000000010002A6A4 C7 45 C0 00 00 00 00 mov dword ptr ss:[rbp-30],0
0000000010002A6B1 48 8B 45 D0 mov rax,qword ptr ss:[rbp-30]
0000000010002A6B5 48 85 C0 test rax,rax
0000000010002A6B8 74 08 je ctf2.10002A6C2
0000000010002A6BA 48 8B 40 F8 mov rax,qword ptr ds:[rax-8]
0000000010002A6BE 48 83 C0 01 add rax,1
0000000010002A6C2 48 8D 48 FF lea rcx,qword ptr ds:[rax-1]
0000000010002A6C6 C7 45 C8 00 00 00 00 mov dword ptr ss:[rbp-38],0
0000000010002A6C9 3B 4D C8 cmp ecx,dword ptr ss:[rbp-38]
0000000010002A6D0 7C 67 jn ctf2.10002A739
0000000010002A6D2 83 6D C8 01 sub dword ptr ss:[rbp-38],1
0000000010002A6D6 66 90 nop
0000000010002A6D8 83 45 C8 01 add dword ptr ss:[rbp-38],1
0000000010002A6DC 48 63 45 C0 movsxd rax,dword ptr ss:[rbp-40]
0000000010002A6E0 48 99 cqo
0000000010002A6E2 49 88 02 00 00 00 00 movabs r8,2
0000000010002A6EC 49 F7 F8 idiv r8
0000000010002A6EF 48 85 D2 test rdx,rdx
0000000010002A6F2 75 1F jne ctf2.10002A713
0000000010002A6F4 48 8B 55 D0 mov rdx,qword ptr ss:[rbp-30]
0000000010002A6F8 48 63 45 C8 movsxd rax,dword ptr ss:[rbp-38]
0000000010002A700 0F 86 14 02 movzx edx,byte ptr ds:[rdx+rax]
0000000010002A703 8B 45 F0 mov eax,dword ptr ss:[rbp-10]
0000000010002A705 31 D0 xor eax,edx
0000000010002A709 4C 8B 45 D0 mov r8,qword ptr ss:[rbp-30]
0000000010002A70C 48 63 55 C8 movsxd rdx,dword ptr ss:[rbp-38]
0000000010002A710 41 88 04 10 mov byte ptr ds:[r8+rdx],al
0000000010002A713 EB 1D jmp ctf2.10002A730
0000000010002A716 48 8B 45 D0 mov rax,qword ptr ss:[rbp-30]
0000000010002A719 48 63 55 C8 movsxd rdx,dword ptr ss:[rbp-38]
0000000010002A71C 0F 86 14 10 movzx edx,byte ptr ds:[rax+rdx]
0000000010002A71F 8B 45 E8 mov eax,dword ptr ss:[rbp-18]
0000000010002A722 31 D0 xor eax,edx
0000000010002A724 48 8B 55 D0 mov rdx,qword ptr ss:[rbp-30]
0000000010002A728 4C 63 45 C8 movsxd r8,dword ptr ss:[rbp-38]
0000000010002A72C 42 88 04 02 mov byte ptr ds:[r8+rdx],al
0000000010002A730 83 45 C0 01 add dword ptr ss:[rbp-40],1
0000000010002A734 3B 4D C8 cmp ecx,dword ptr ss:[rbp-38]
0000000010002A737 7F 9F ja ctf2.10002A6D8
0000000010002A739 4C 8D 05 90 89 19 00 lea r8,qword ptr ds:[1001C30D0]
0000000010002A740 48 8D 00 31 D4 19 00 lea rcx,qword ptr ds:[1001C7878]
0000000010002A747 41 89 00 FF 00 00 mov r3d,0
0000000010002A74D 48 BA 01 00 00 00 00 movabs rdx,1
0000000010002A757 F8 44 F6 00 00 call ctf2.10003D0A0

```



However this doesn't bring us any closer to figuring out the password on the ZIP archive. Trying the following passwords brings no luck:

WAR\*@\*IT\*FEST\*2020

binary.txt

binary.zip

IT\*FEST rulz!

sour

c54e

C54E

4ec5

4EC5

- all of the above prepended with "sour"
- all numbers from 0 to 65535