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 teeest2.txt
abcdefghijklmnopqrstuvwxyz0123945789abcdefghijklmnopqrstuvwxyz0123945789
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ ctf1 -c teeest2.txt
OTo7PDO+PzAxMjMONTY3KCkqKywtLi8gISJoaWprYWxtb2BhOTo7PDO+PzAxMjMONTY3KCkqKywtLi8gISJoaWprYWxtb2Bh
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ ctf1 -c teeest2.txt | base64 -d
9:;<=>?01234567()*+,-./ !"hijkalmo`a9:;<=>?01234567()*+,-./ !"hijkalmo`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.

```python
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.

```python
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, 9, 18, 127, 121]
abcdefgh -> Eg0WFhYJEhp+ -> [18, 13, 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 “aaaaaaaa” 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`:

```python
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:

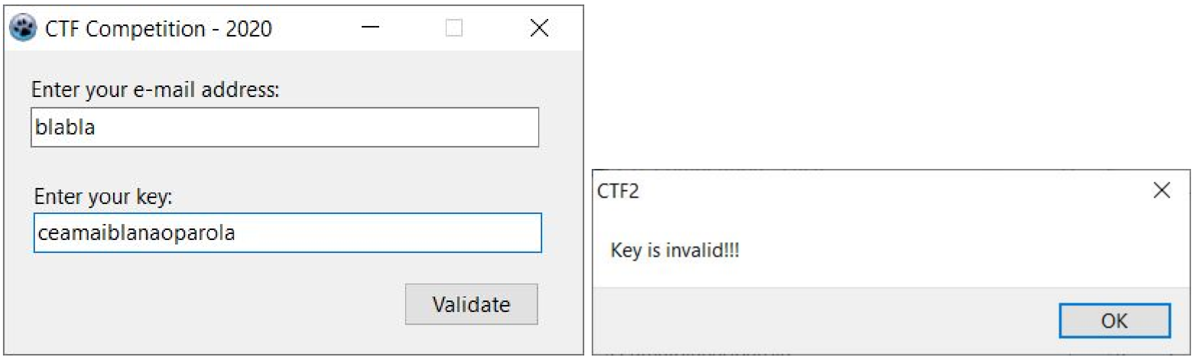
```
cvijdea@CVIJDEA-L2 C:\Projects\itfest-ctf
$ python x.py binary3.dat
Amazing work!!!    Your nuclear codes are: WAR*@*IT*FEST*2020
```

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

PROBLEM 2

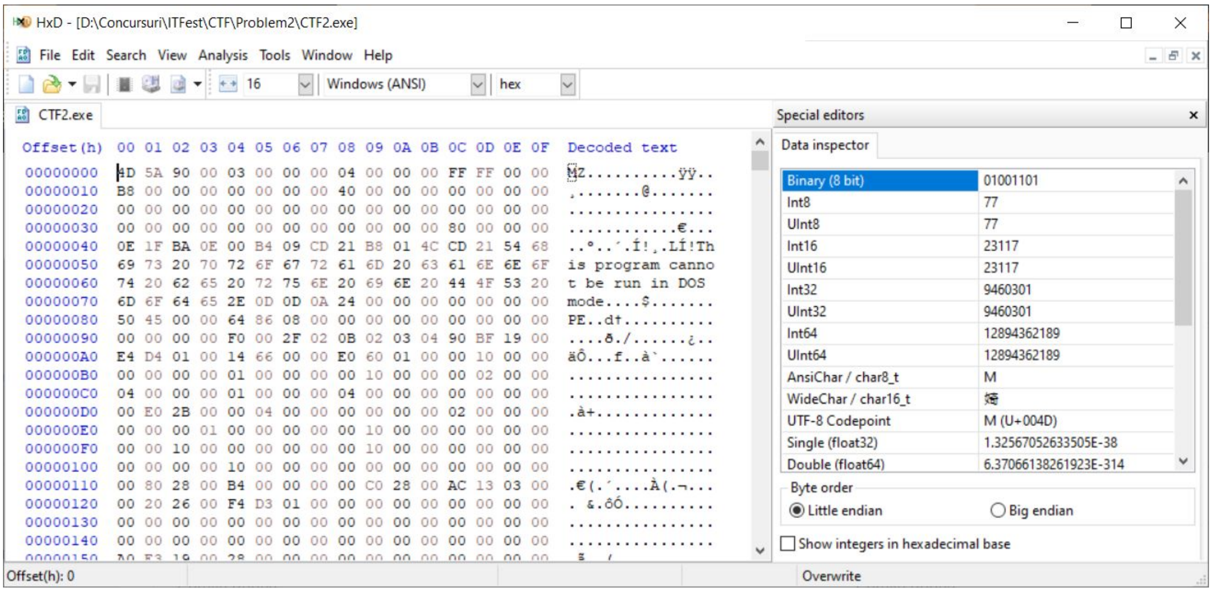
MISSION 1

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

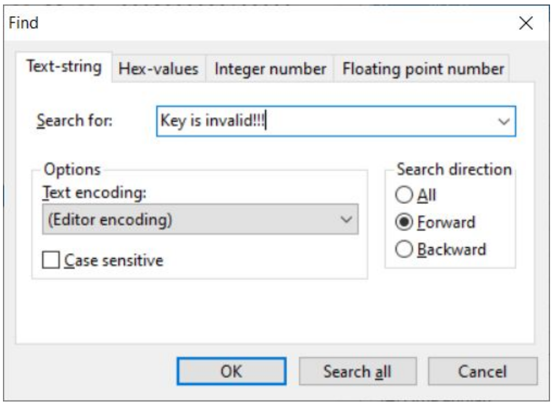


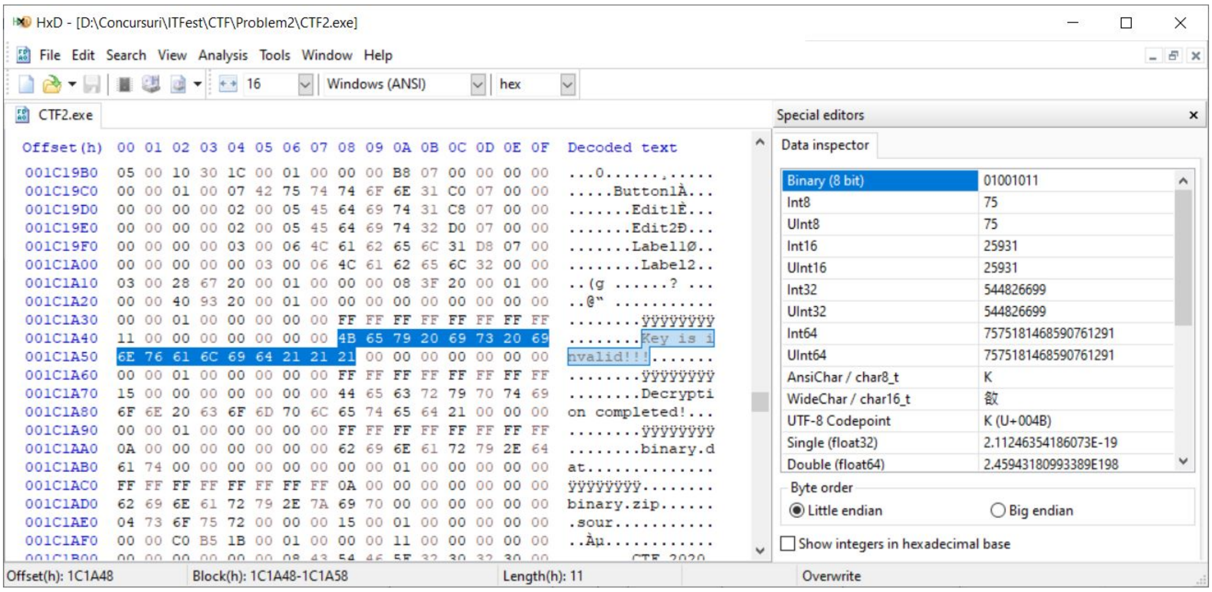
We notice that the string *"Key is invalid!!!"* is the string that needs to be converted to *"IT*FEST rulz!"*.

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

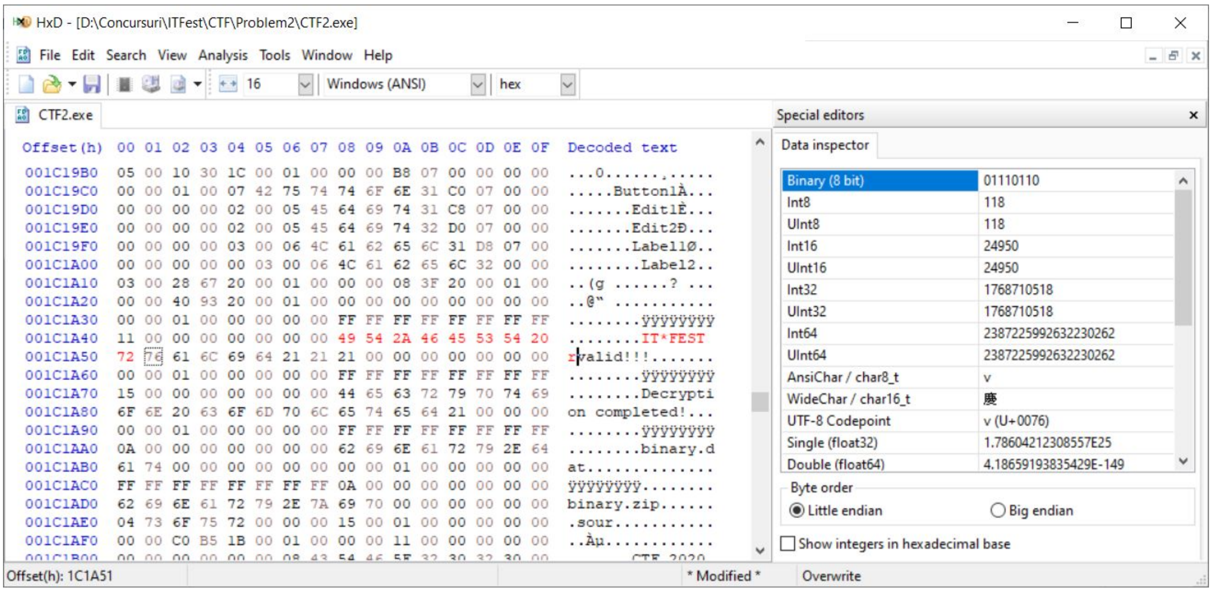


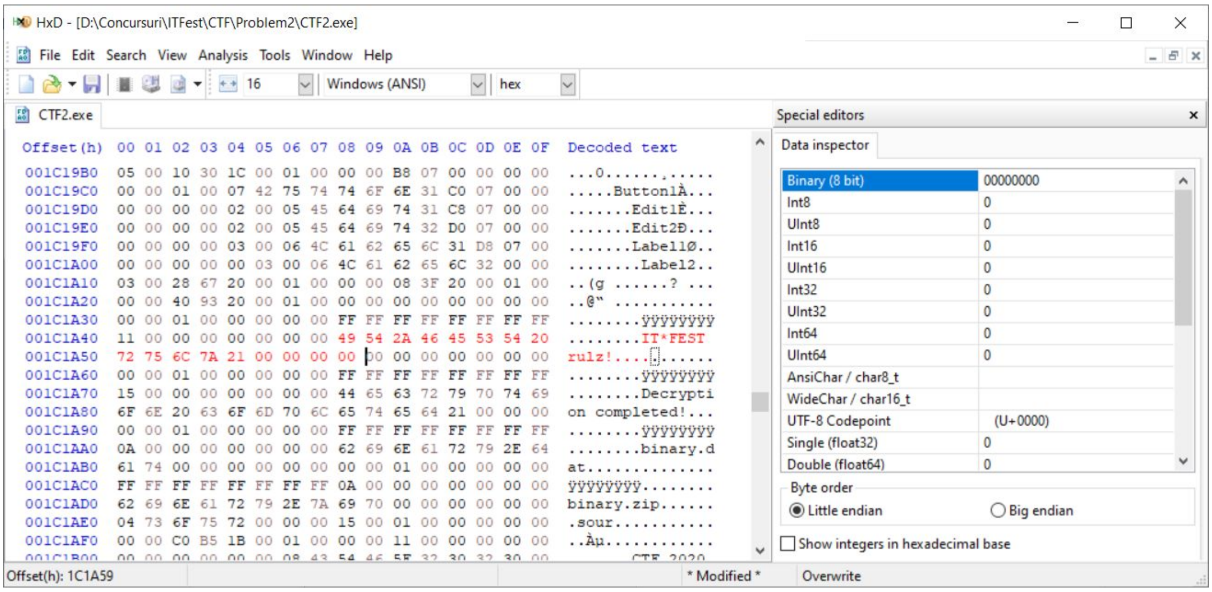
Now that we know the message we search for the string *"Key is invalid!!!"*.



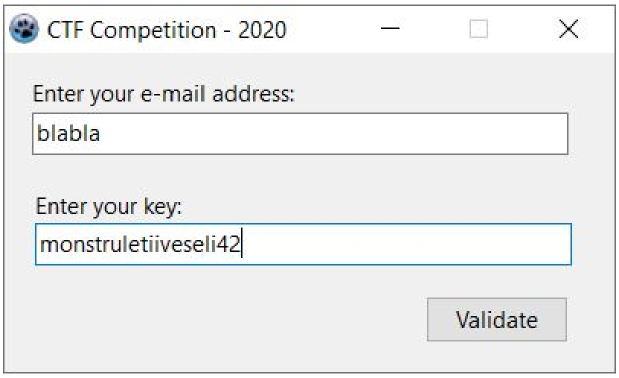


We replace *"Key is invalid!!!"* with *"IT*FEST rulz!"* and the remaining characters (because *"Key is invalid!!!"* has 17 characters and *"IT*FEST rulz!"* has only 13 characters) with the hex value `0x00` and we save the file (as `CTF2_mission1.exe`).





We run `CTF2_mission1.exe` and give random strings as inputs for both email and password to see if we got it right.

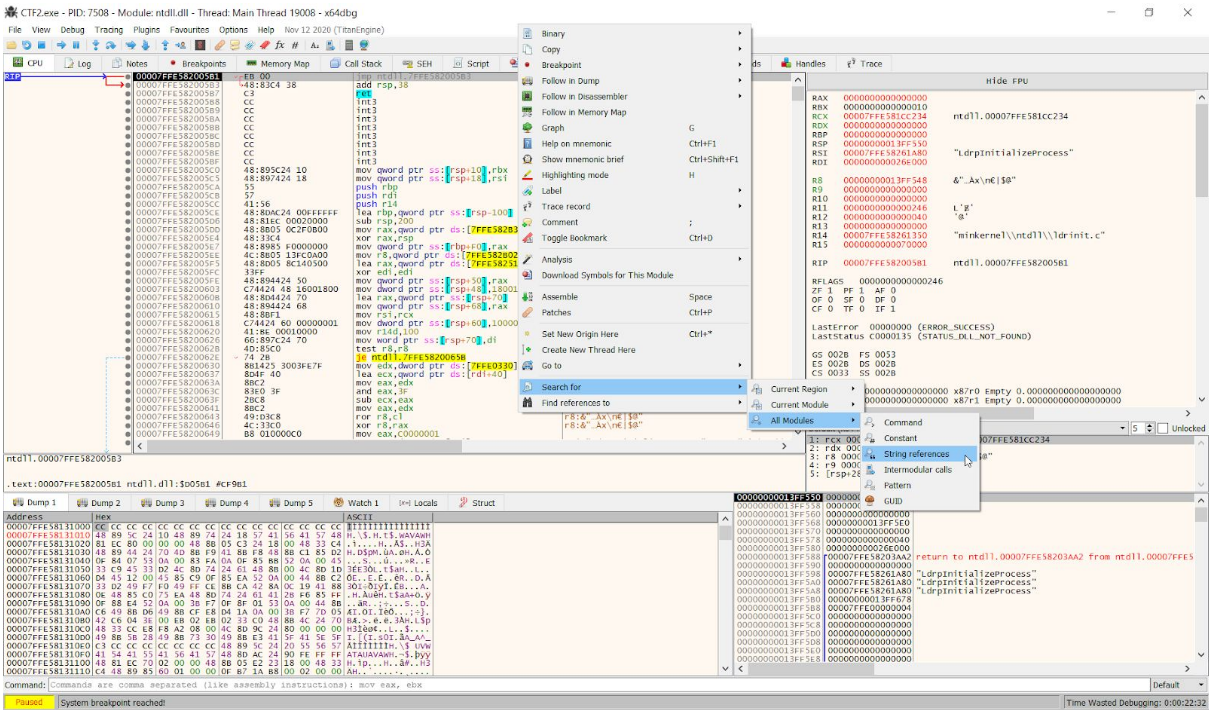


Yay! It worked!

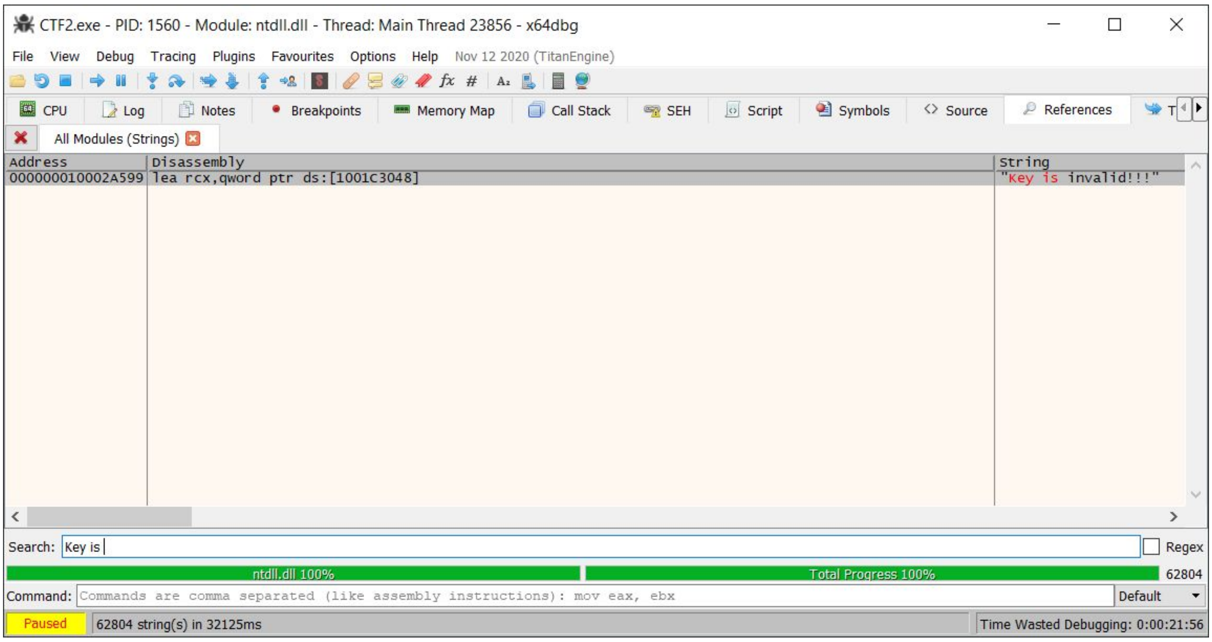
Solution: `CTF2_mission1.exe` (you can find our modified file in the same folder as this writeup)

MISSION 2

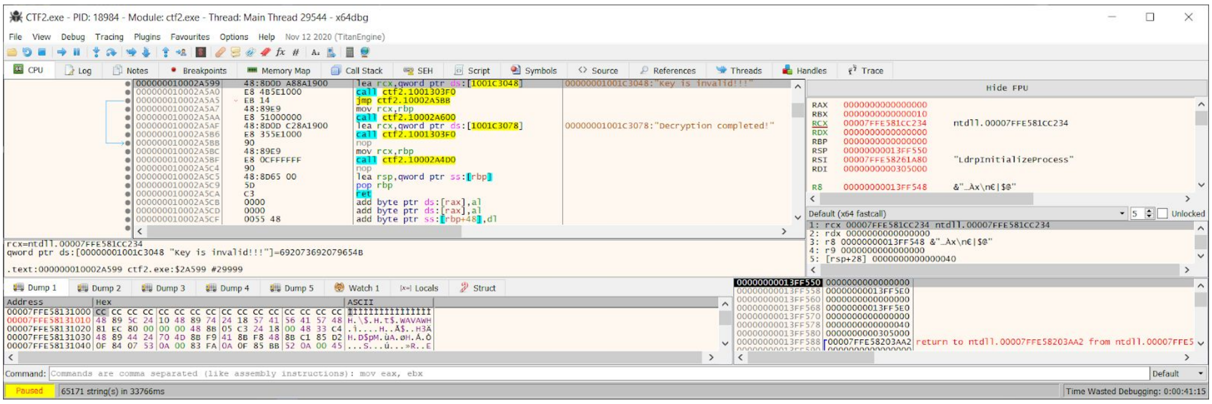
We open `CTF2.exe` using the program x64dbg which is an open-source x64/x32 debugger for windows.



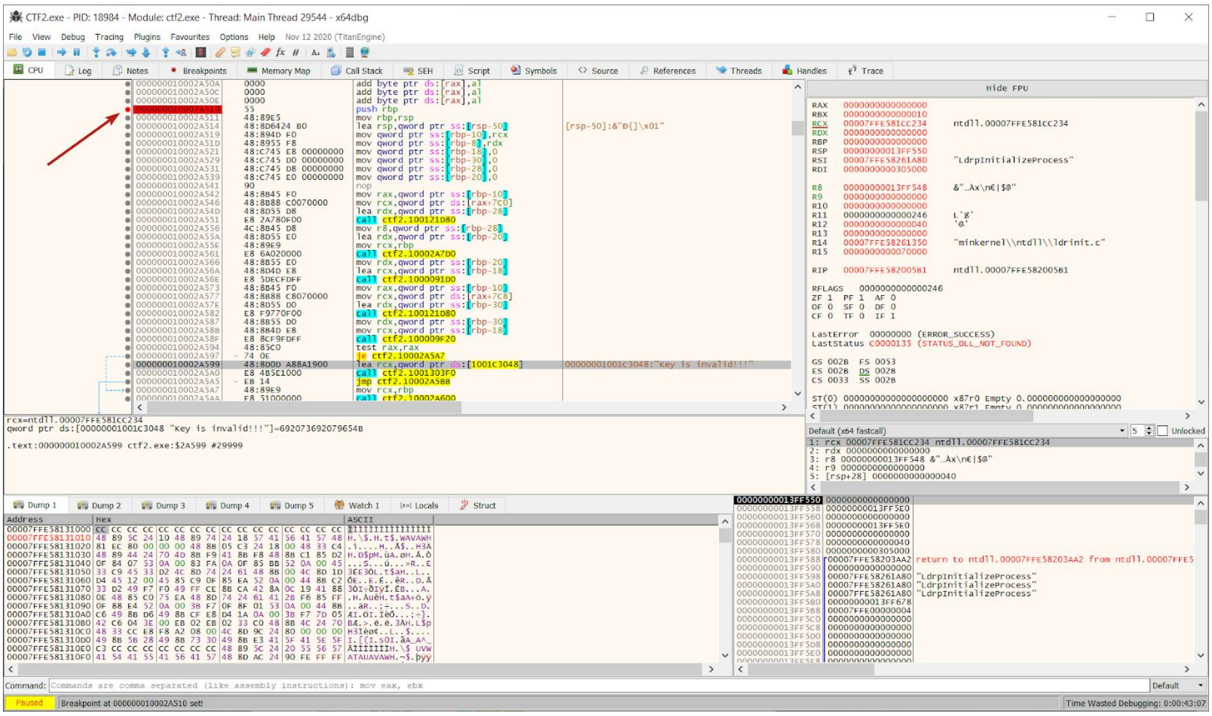
We search for string references, we get the list with all the strings contained by the executable and we search for *"Key is invalid!!!"*.



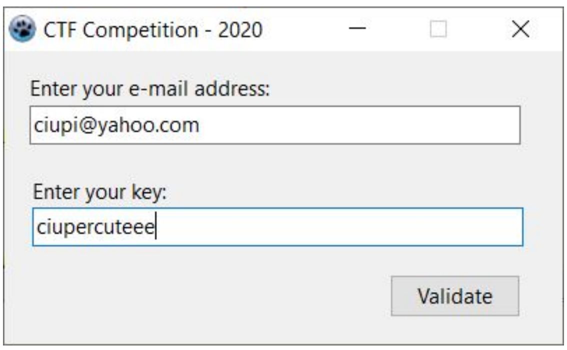
We double click our search result to see where it takes us.



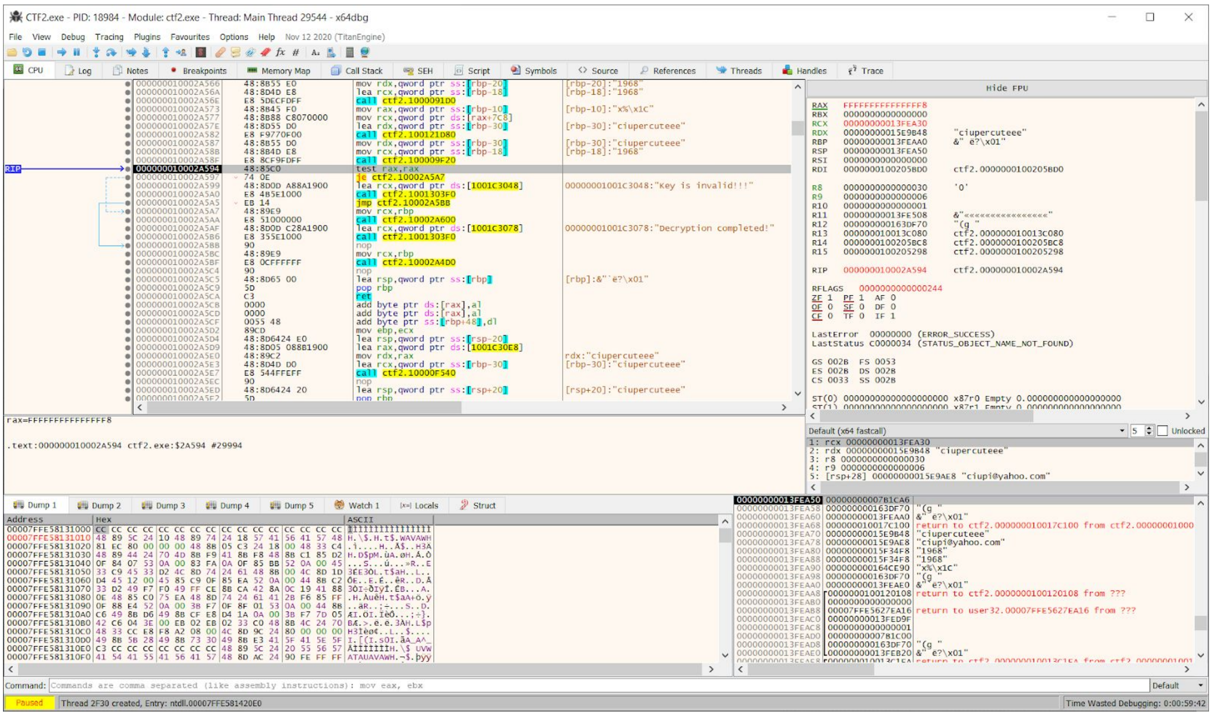
We set a breakpoint at the beginning of the function which checks our password.



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



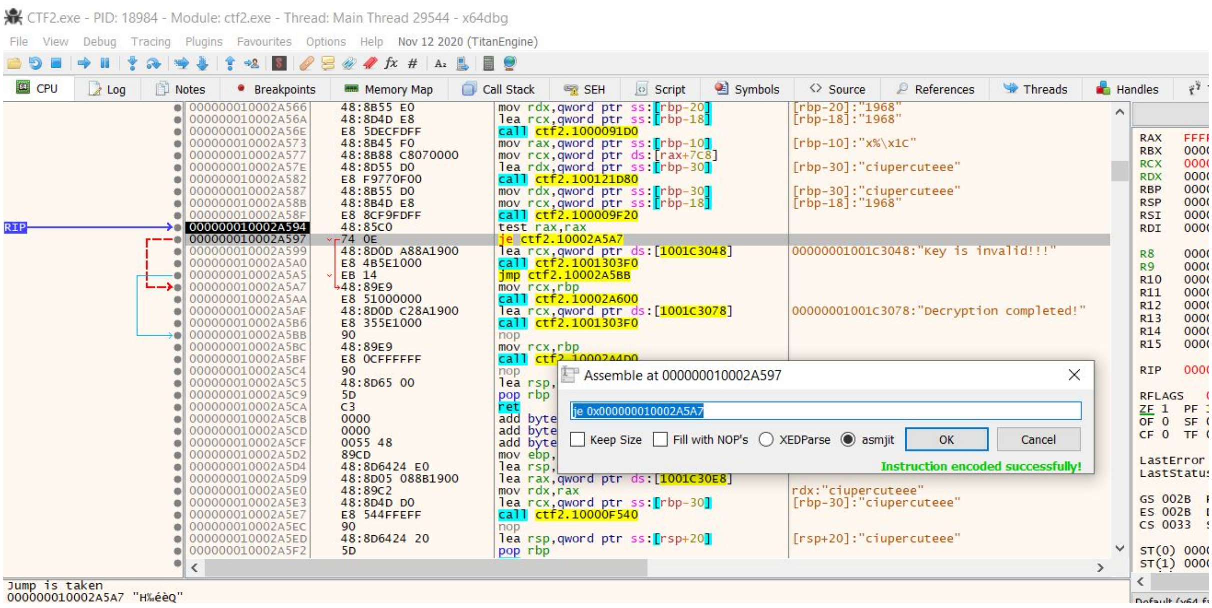
We reach our breakpoint and start running the program step by step using Step into (F7) and Step over (F8) in order to understand what it does.

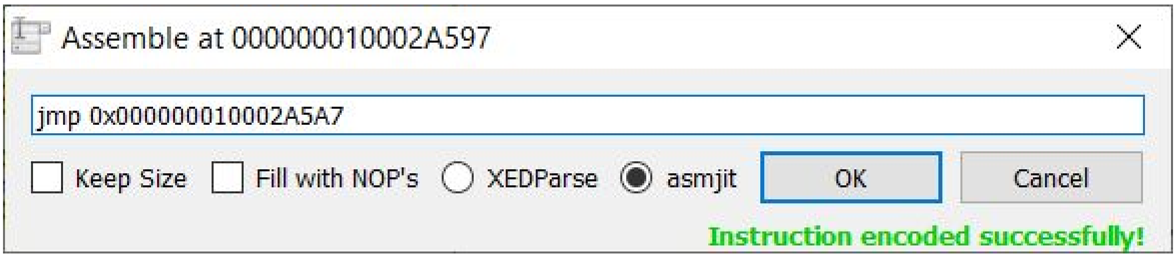


As we can see, if the two values that are being compared are equal, we get the message *"Decryption completed!"*, otherwise we get the message *"Key is invalid!!!"*.

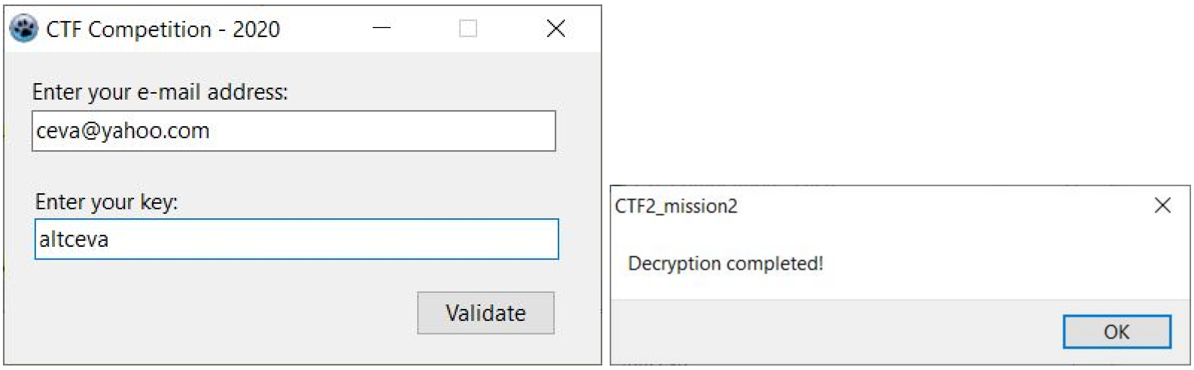
Using the built in assembler, we can change the jump instruction. That means that if we change it to `jmp` instead of `je`, it will always show that the password is valid regardless of what we enter.

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.

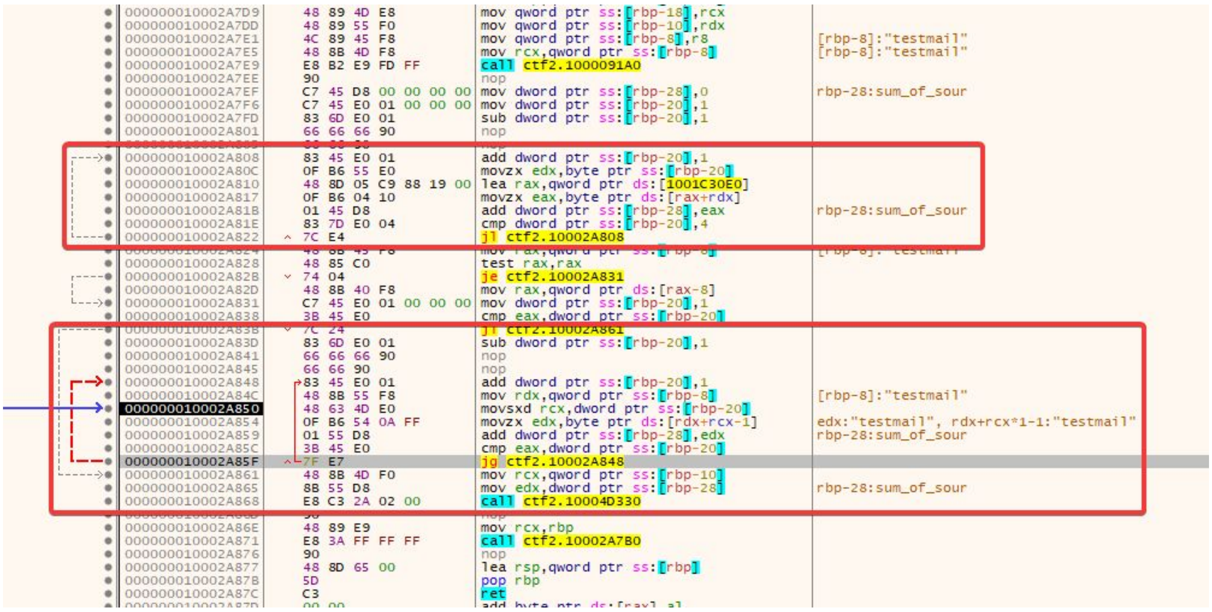


The message we get after clicking Validate is “Decryption completed!”. We also notice a zip file named `binary.zip` appears in our folder, which is a password-protected ZIP archive containing `binary.txt` inside it.

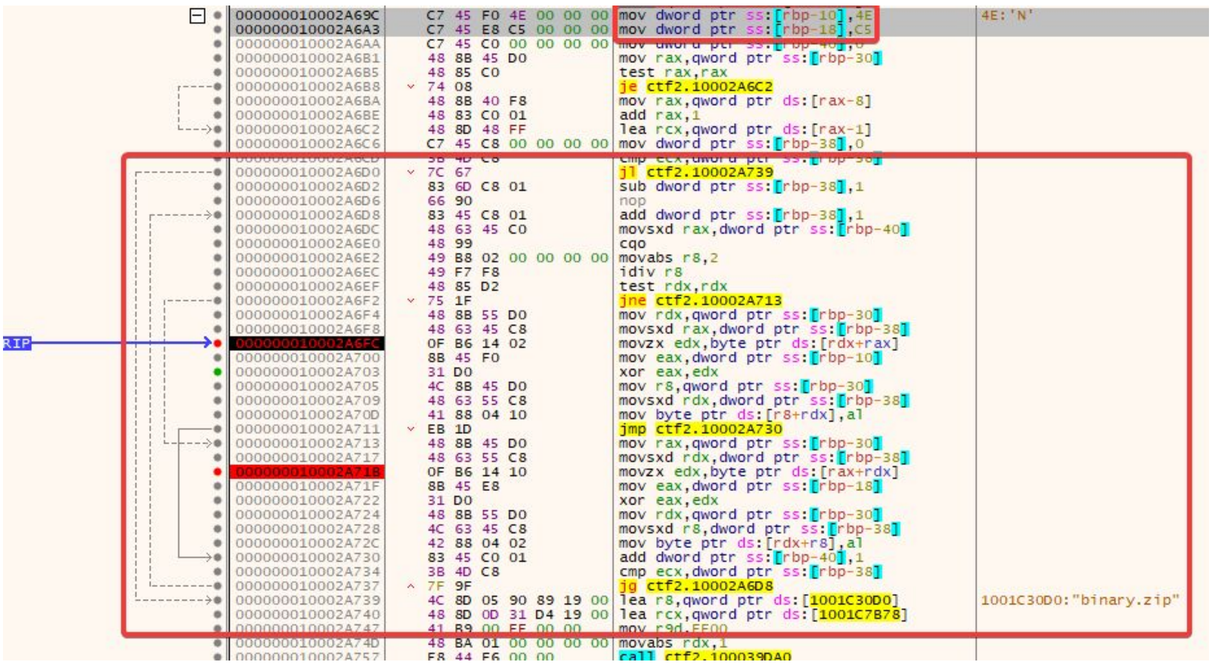
Solution: `CTF2_mission2.exe`

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



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`.



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