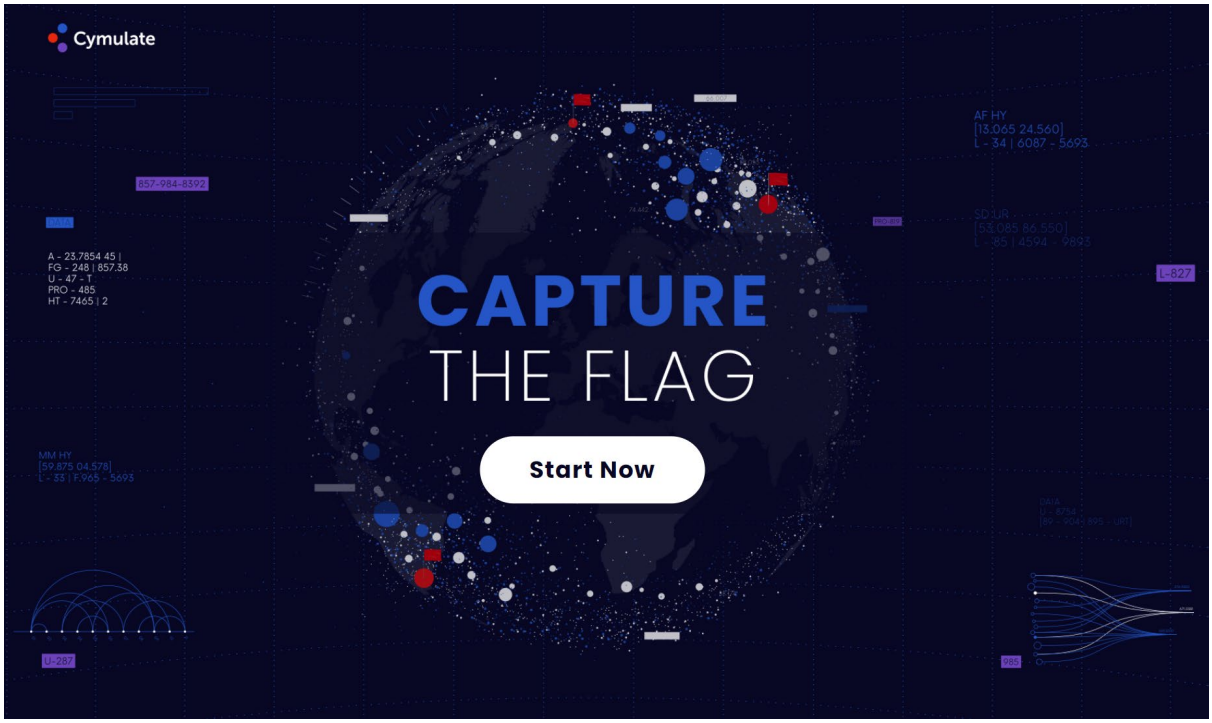
Cymulate CTF – Black Hat 2022

Introduction

In this write-up, I will present the solution to the reversing challenge called ‘Binushka’ from the Cymulate CTF event.
The challenge was published in 1.8.2022 and was designed for the Black Hat 2022 event.
Every candidate who solved that challenge could claim his prize at Cymulate Black Hat boot #1874.



Binushka challenge (Reversing)

The first thing we should look for when we come to solve a CTF challenge is the name of the challenge. The name is '**Binushka'**, We can split the name into two parts: '**Bin'** + '**ushka'.**
The first part of the name is clear.
'**Bin'** -> Binary, we can also see that the name of the file is '**bin_bin_bin'**.
About the second part, '**ushka'** if we are creative in our mind we maybe can think about babushka.
Apart from the literal meaning of babushka (grandmother in Russian), there is also a doll named '**Babushka doll'** or in the official name **'Matryoshka doll'.**
Matryoshka doll is a set of wooden dolls of decreasing size placed one inside another.
We can also think that maybe the file name '**bin_bin_bin'** is a hint that there are some binaries inside binaries like Matryoshka doll of binaries.
'bin_bin_bin' -> 'bin_bin' -> 'bin'.

Now, after we dedicated some time to thinking about the challenge and the file name, we can download the challenge and start reversing the binary.
The first thing we will do is to run the binary and see what happens.
It seems like the binary tries to open a file that doesn't exist.

```
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ ./bin_bin_bin
Error opening file.: No such file or directory
```

Let's open the file in IDA.

Pseudocode-A

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    FILE *stream; // [rsp+Ch] [rbp-13348h]
4    char v5; // [rsp+354h] [rbp-13000h] BYREF
5    __int64 v6[512]; // [rsp+12354h] [rbp-1000h] BYREF
6
7    while ( v6 != (__int64 *)&v5 )
8      ;
9    v6[511] = __readfsqword(0x28u);
10   if ( (unsigned int)antidebug() )
11     return 1;
12   stream = fopen("bin_bin", "r");
13   if ( stream )
14   {
15     fseek(stream, 0LL, 2);
16     ftell(stream);
17     fseek(stream, 0LL, 0);
18     puts("It seems like you've missed something...");
19   }
20   else
21   {
22     perror("Error opening file.");
23   }
24   return -1;
25 }
```

Pseudocode-A

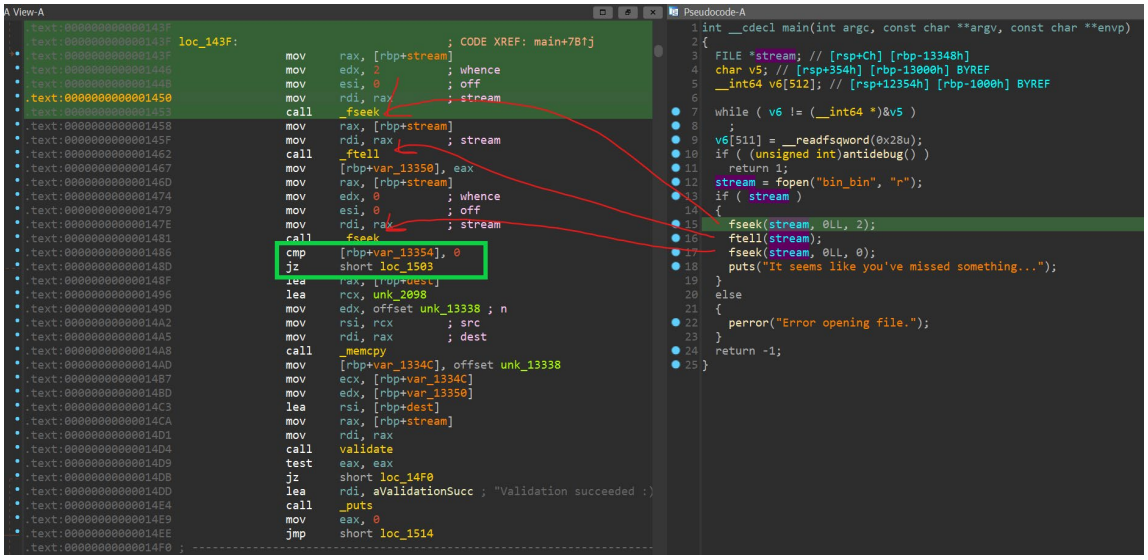
```
1  __int64 antidebug()
2  {
3    __int64 result; // rax
4
5    result = ptrace(PTRACE_TRACEME, 0LL, 1LL, 0LL);
6    if ( result == -1 )
7    {
8      puts("I'm being debugged!");
9      return 1LL;
10   }
11   return result;
12 }
```

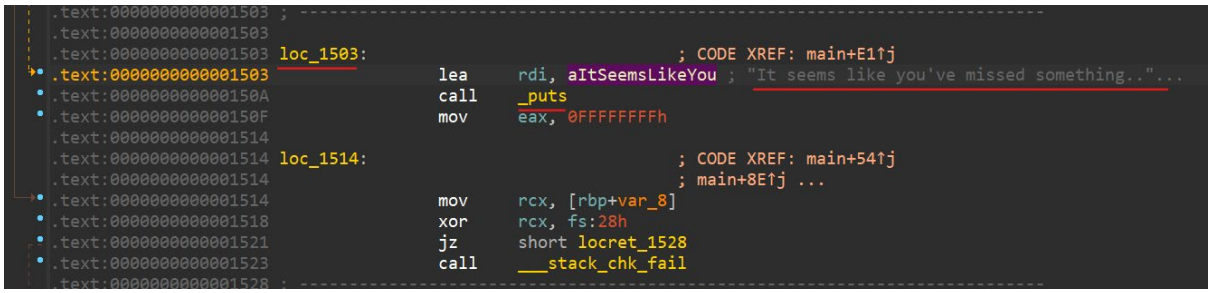
First, We can see a call to a function called '**antidebug'**, and as you can guess this function checks if the binary is debugged with the '**ptrace'** command. If the answer is yes the process is terminated.
We can also see an attempt to open a file named '**bin_bin**' and if the file doesn't exist we got the error we saw above.
Now we will create a file named **'bin_bin**' and see what's happening.

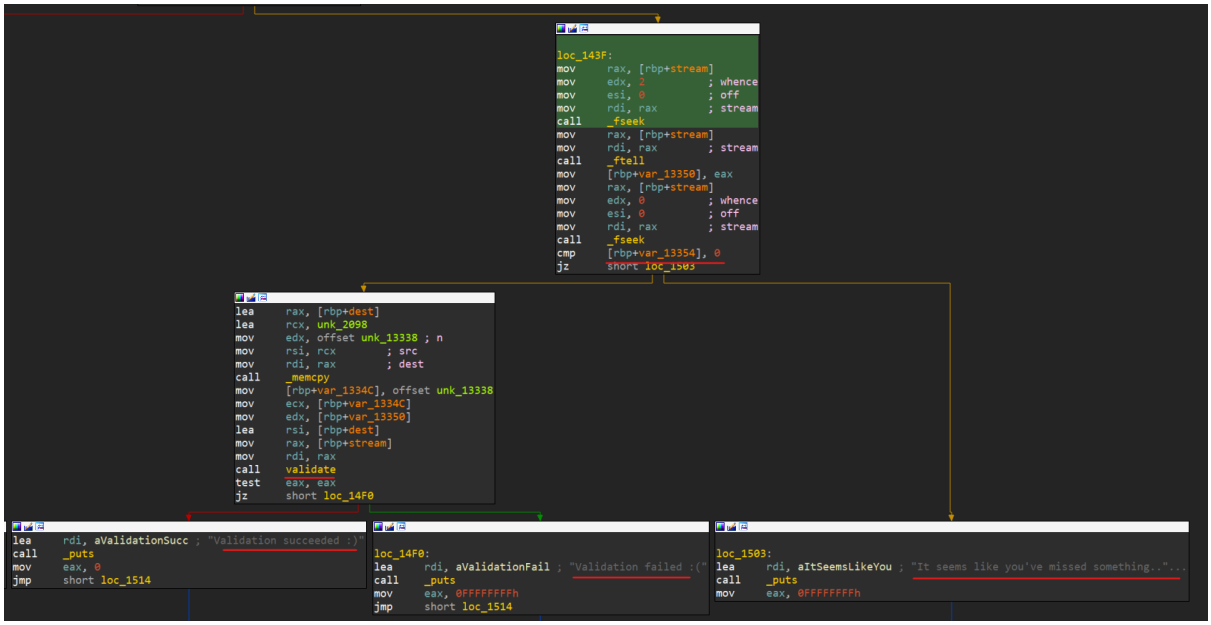
```
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ touch bin_bin
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ ./bin_bin_bin
It seems like you've missed something...
```

It seems we hit the **'puts'** function we saw above after we entered the if condition.
So, apart from calculating the size of the file with the **'ftell'** function what else happening here?
In this case, we need to check the assembly code to see the whole picture.

As we can see, after the last **'fseek'** function there is **'cmp'** call to a variable on the stack with the value of zero. If the variable is zero, we jump to '**loc_1503**' and this leads us to the **'puts'** that we saw above.

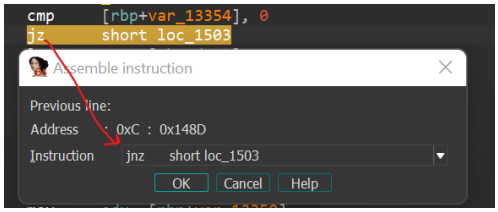






If the value is **not zero,** we are jumping to another flow that contains a **validate** function.
(The IDA decompiler didn't show this at all, because the variable value **is hard coded zero** so there is no scenario that we go to the second flow…), As long as we don't patch the binary 😊

First, let's patch the binary and change the **'jz'** to **'jnz'** which means we will not jump to '**loc_1503**'.



After we patched the file and ran the binary, we got the output **'Validation failed ☹'.**

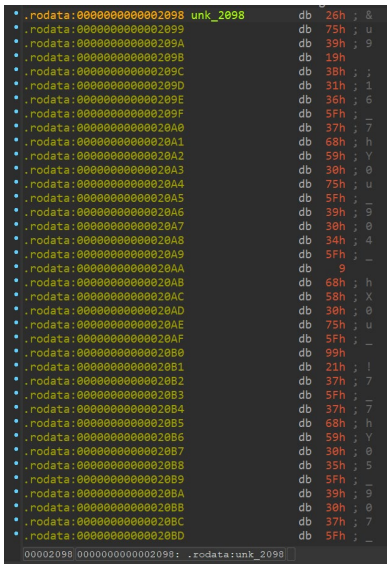
```
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ ./bin_bin_bin
Validation failed :(
```

Let's return to the second flow where we saw the **'validate'** function.
As we can see, Now IDA decompiler shows us the flow of the **'validate'** function, due to our patch.

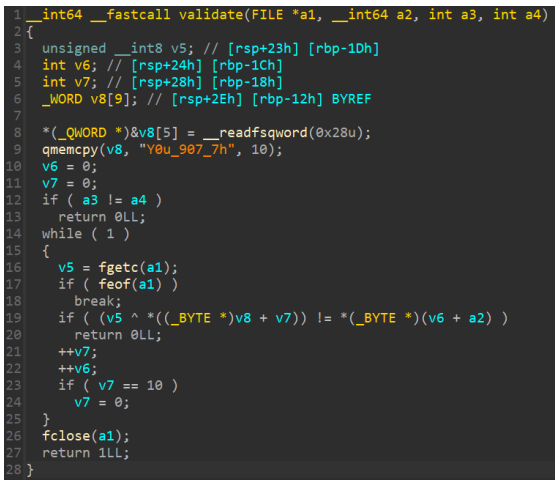
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  unsigned int v4; // [rsp+4h] [rbp-13350h]
  FILE *stream; // [rsp+Ch] [rbp-13348h]
  char dest[832]; // [rsp+14h] [rbp-13340h] BYREF
  char v7; // [rsp+354h] [rbp-13000h] BYREF
  __int64 v8[512]; // [rsp+12354h] [rbp-1000h] BYREF

  while ( v8 != (__int64 *)&v7 )
    ;
  v8[511] = __readfsqword(0x28u);
  if ( (unsigned int)antidebug(argc, argv, envp) )
    return 1;
  stream = fopen("bin_bin", "r");
  if ( stream )
  {
    fseek(stream, 0LL, 2);
    v4 = ftell(stream);  v4 = size of 'bin_bin' file.
    fseek(stream, 0LL, 0);
    memcpy(dest, &unk_2098, (size_t)&unk_13338);    0x13338
    if ( (unsigned int)validate(stream, dest, v4, &unk_13338) )
    {
      puts("Validation succeeded :)");
      return 0;
    }
    else
    {
      puts("Validation failed :(");
      return -1;
    }
  }
  else
  {
    perror("Error opening file.");
    return -1;
  }
}
```

So, as you assume our target is to hit the '**Validation succeeded ☺**' which means we created the correct **'bin_bin'** file.
To do so, we need that the **'validate'** function will return **true**.
<u>Variables:</u>
1. The variable 'v4' will store the size of the desired 'bin_bin' file
2. The variable 'dest' will contain a stream of bytes in the size of 0x13338.
3. As we can see the 'validate' function will get four arguments.
    - File descriptor to our 'bin_bin' file.
    - 'dest' variable which contains a stream of bytes in the size of 0x13338.
    - Size of 'bin_bin' file.
    - Value of 0x13338

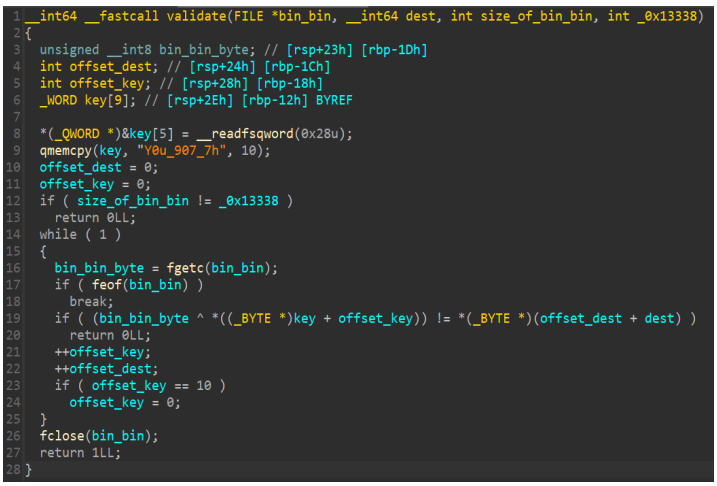
The stream of bytes in size of 0x13338 that will be copied to the ‘dest’ variable:



Now let’s dive into the validate function and make some refactoring:



```
__int64 __fastcall validate(FILE *a1, __int64 a2, int a3, int a4)
{
  unsigned __int8 v5; // [rsp+23h] [rbp-1Dh]
  int v6; // [rsp+24h] [rbp-1Ch]
  int v7; // [rsp+28h] [rbp-18h]
  _WORD v8[9]; // [rsp+2Eh] [rbp-12h] BYREF

  *(_QWORD *)&v8[5] = __readfsqword(0x28u);
  qmemcpy(v8, "Y0u_907_7h", 10);
  v6 = 0;
  v7 = 0;
  if ( a3 != a4 )
    return 0LL;
  while ( 1 )
  {
    v5 = fgetc(a1);
    if ( feof(a1) )
      break;
    if ( (v5 ^ *((_BYTE *)v8 + v7)) != *(_BYTE *)(v6 + a2) )
      return 0LL;
    ++v7;
    ++v6;
    if ( v7 == 10 )
      v7 = 0;
  }
  fclose(a1);
  return 1LL;
}
```



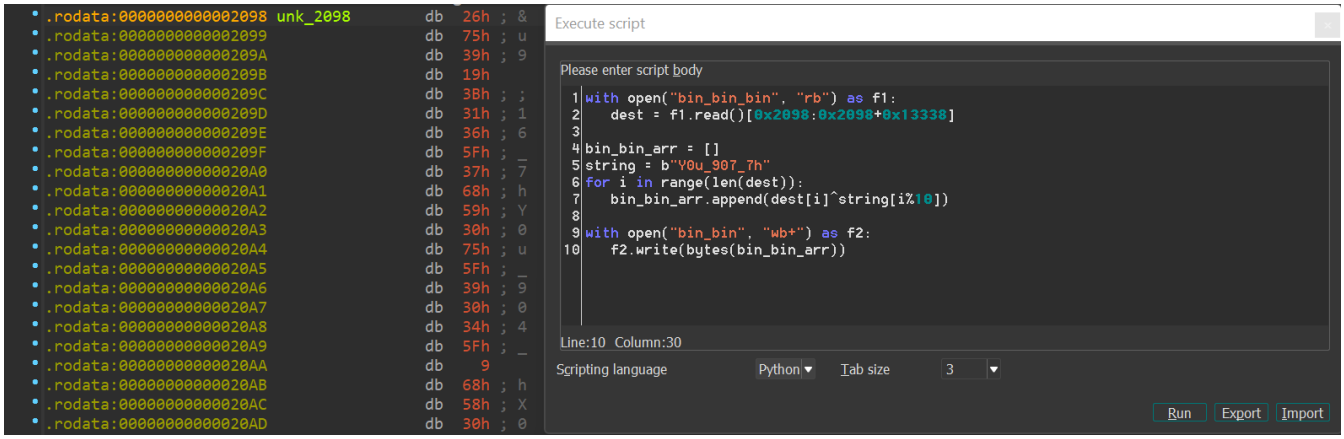
```
__int64 __fastcall validate(FILE *bin_bin, __int64 dest, int size_of_bin_bin, int _0x13338)
{
  unsigned __int8 bin_bin_byte; // [rsp+23h] [rbp-1Dh]
  int offset_dest; // [rsp+24h] [rbp-1Ch]
  int offset_key; // [rsp+28h] [rbp-18h]
  _WORD key[9]; // [rsp+2Eh] [rbp-12h] BYREF

  *(_QWORD *)&key[5] = __readfsqword(0x28u);
  qmemcpy(key, "Y0u_907_7h", 10);
  offset_dest = 0;
  offset_key = 0;
  if ( size_of_bin_bin != _0x13338 )
    return 0LL;
  while ( 1 )
  {
    bin_bin_byte = fgetc(bin_bin);
    if ( feof(bin_bin) )
      break;
    if ( (bin_bin_byte ^ *((_BYTE *)key + offset_key)) != *(_BYTE *)(offset_dest + dest) )
      return 0LL;
    ++offset_key;
    ++offset_dest;
    if ( offset_key == 10 )
      offset_key = 0;
  }
  fclose(bin_bin);
  return 1LL;
}
```

We can notice an interesting string, ‘**Y0u_907_7h’.**

If we analyze the code, it looks like there is a **xor** operation between the **‘bin_bin’** file and the string **‘Y0u_907_7h’**, and every result of the xor operation will compare to the byte stream that is stored in dest.

If we will encounter a byte from ‘bin_bin’ that after the xor operation does not equal to the equivalent byte in the dest variable, the function will return zero (False) ☹

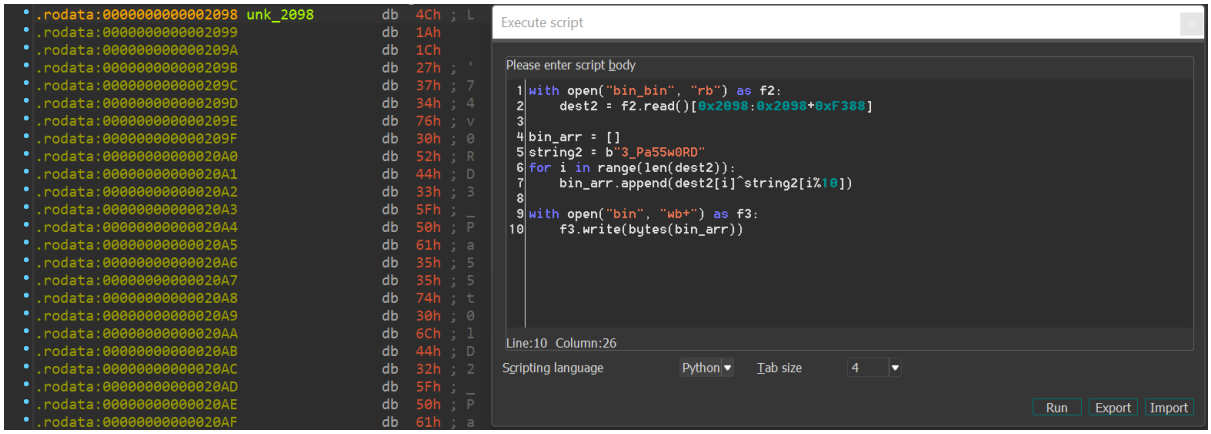
So, our mission is clear, to **generate a valid 'bin_bin'** file we need to execute the xor operation between the string and the stream of bytes in the 'dest' variable.
We will execute a python script in IDA.
Offset 0x2098 is the start of the byte stream that is copied to our 'dest' variable, in size 0x13338.



And…what a surprise! our 'bin_bin' file is an ELF file, and the validation was succeeded.

```
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ file bin_bin
bin_bin: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d4d0dae3508b1de8264f9e2c70417dc55f99583
0, for GNU/Linux 3.2.0, not stripped
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ ./bin_bin_bin
Validation succeeded :)
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$
```

Long story short, the 'bin_bin' file has the same logic as 'bin_bin_bin', the only difference is the string which used for the xor operation, the string is **'3_Pa55w0RD'.**
If we will concatenate the two strings, we will get the string: '**Y0u_907_7h3_Pa55w0RD**'.



And as you can see 'bin' is an ELF file and the validation succeeded.

```
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ file bin
bin: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e028f983389db88a994924846bba7372b91a4674, f
or GNU/Linux 3.2.0, not stripped
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$ ./bin_bin
Validation succeeded :)
elad@eladbeber:/mnt/c/Users/User/Desktop/Cymulate$
```

The last binary is a little different, let's open it in IDA after we patch the 'jz' command to 'jnz', like in the previous binaries.

```
// bad sp value at call has been detected, the output may be wrong!
int __cdecl main(int argc, const char **argv, const char **envp)
{
  unsigned int size; // [rsp+18h] [rbp-B458h]
  FILE *stream; // [rsp+20h] [rbp-B450h]
  void *ptr; // [rsp+28h] [rbp-B448h]
  char dest[1088]; // [rsp+30h] [rbp-B440h] BYREF
  char v8; // [rsp+470h] [rbp-B000h] BYREF
  __int64 v9[512]; // [rsp+A470h] [rbp-1000h] BYREF

  while ( v9 != (__int64 *)&v8 )
    ;
  v9[511] = __readfsqword(0x28u);
  if ( (unsigned int)antidebug(argc, argv, envp) )
    return -1;
  stream = fopen("flag.zip", "r");
  if ( stream )
  {
    fseek(stream, 0LL, 2);
    size = ftell(stream);
    fseek(stream, 0LL, 0);
    ptr = malloc(size);
    fread(ptr, size, 1uLL, stream);
    shuffle(ptr, size);
    memcpy(dest, &unk_2098, 0xB431uLL);
    if ( (unsigned int)validate((__int64)ptr, (__int64)dest, size, 46129) )
    {
      puts("Validation succeeded :)");
      return 0;
    }
    else
    {
      puts("Validation failed :(");
      return -1;
    }
  }
  else
  {
    perror("Error opening file.");
    return -1;
  }
}
```

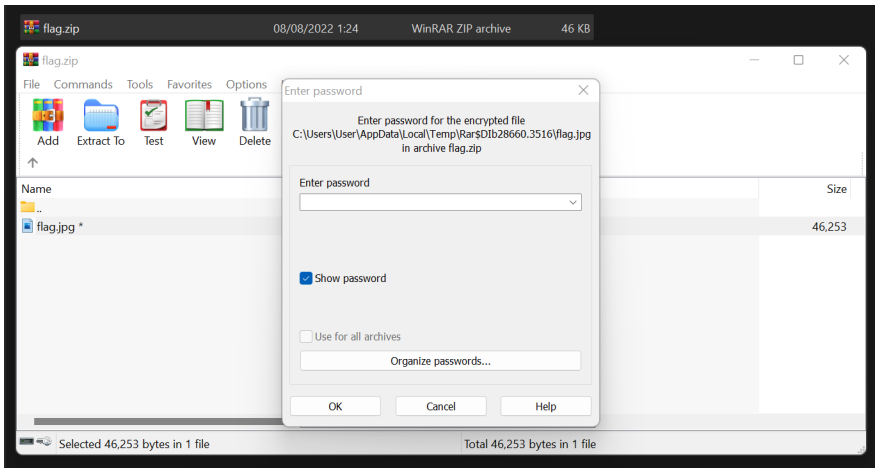
In this binary, instead of **xor** operation, we have the function **'shuffle'**, and this time the shuffle will be executed on the file named **'flag.zip'** (seems like we are close to the flag).
The shuffle accepts two arguments, the desired file **('flag.zip')** and his size.
Basically, the function initiates the rand function with the seed **0x1337**.
After that, generates random numbers that will be used as indexes for swap operations.
So, the program opens the file and **shuffles all its bytes, and later compares the shuffled file with the 'dest' variable in the validate function**.

```
__int64 __fastcall shuffle(__int64 a1, int a2)
{
  __int64 result; // rax
  int v3; // eax
  int i; // [rsp+18h] [rbp-8h]

  srand(0x1337u);
  result = (unsigned int)(a2 - 1);
  for ( i = a2 - 1; i > 0; --i )
  {
    v3 = rand();
    result = swap(i + a1, a1 + v3 % (i + 1));
  }
  return result;
}
```

So, our mission now is to build an ‘**unshuffle’** script that will be executed on the **‘dest’** variable to get the correct **‘flag.zip’** file. Due to the fact we got the seed **0x1337**, building the **‘unshuffle’** script will not be a problem.

```python
from ctypes import CDLL

libc = CDLL("libc.so.6")
size = 0xb431
seed = 0x1337

libc.srand(seed)
rands = [libc.rand() for i in range(size)][::-1]

with open("bin", "rb") as f_bin:
    dest = bytearray(f_bin.read()[0x2098:0x2098+size])

for i in range(1, size):
    dest[i], dest[rands[i] % (i + 1)] = dest[rands[i] % (i + 1)], dest[i]

with open("flag.zip", "wb") as f_flag:
    f_flag.write(dest)
```

After we ran the python script we get the final file: ‘flag.zip’.
The zip contains an image called ‘flag.jpg’ but it’s locked with a password.



Remember the strings we used to xor the previous binaries?
After we concatenated them above, we got the string: **‘Y0u_907_7h3_Pa55w0RD’.**
Using this password, the file unlocked successfully and we got the flag 😊