Flare-On 10 Challenge 7: flake

By Mike Hunhoff (@mehunhoff)

Flake is a Python PyQt application that is compiled to native Windows x64 using the open-source Python compiler named Nuitka. The program implements a variant of the classic game, Snake, where players score points by guiding a snake on its quest to consume FLARE logos. The snake moves faster as it eats making it progressively harder, and impossible, to control.

Nuitka is an open source, optimizing Python compiler that "translates Python modules into a C level program that then uses libpython and static C files of its own to execute in the same way as CPython does". The project can execute compiled and uncompiled Python code together while supporting all Python library modules and all extension modules freely. Two versions of Nuitka are available: standard and commercial. Nuitka standard can bundle code, dependencies, and data into a single executable and supports acceleration while Nuitka commercial additionally protects code, data, and outputs. More information can be found in Nuitka's online documentation here.

Nuitka translates even the simplest of Python code into very complex native code. This makes reverse engineering difficult but is a nice side effect for developers who don't want users to have access to their Python source code.

Let's analyze Flake and see what we can learn about reverse engineering Nuitka-compiled files along the way.

Basic Static Analysis: `flake.exe`

We are given three files:

- `flake.exe` ~10 MB
- `demo_conf.txt` <1 KB
- `mail.txt` <1 KB

We view the contents of mail.txt:

```
Subject: need your help...

Oliver Shrub, Gardens Department Head, keeps bragging about his high score for this rip off Snake game called
Flake. I'm pretty sure he found a way to cheat the game because there is no way it's possible to score 10,000
points...I mean the game ships with a sketchy TXT file so there must be something fishy going on here.

Can you look at the game and let me know how I can beat Oliver's high score?

Best,

Nox
Shadows Department Head
```

We need to figure out how to beat Oliver Shrub's high score that Nox claims is impossible to get without cheating. Let's view the contents of the file `demo_conf.txt`:


```
WTOh3Rgz17NjWtTfd33llk9w5ZoCQeOQAmegzwI51ZpPfrjdDjOg3Rgkvd0QM6vPDjOi3Rgj7A==
```


This looks like Base64-encoded data – let's try to decode it using CyberChef:

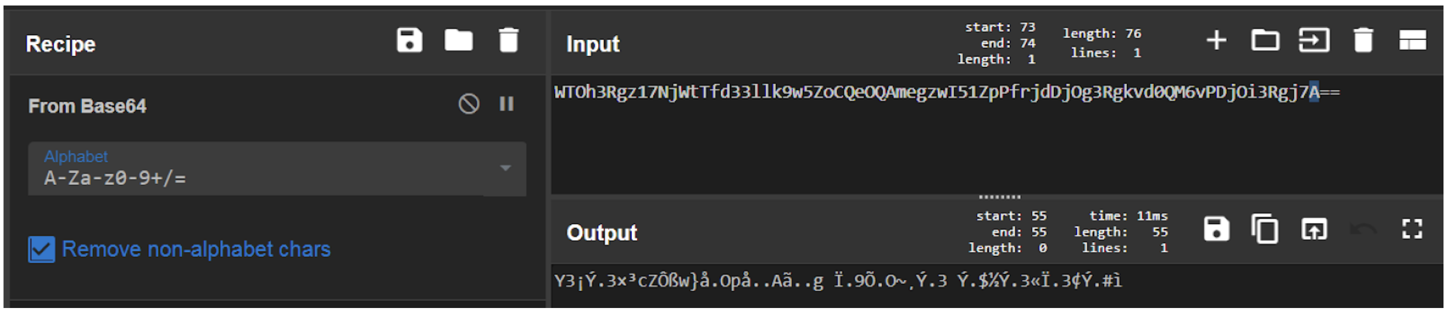

Figure 1: Decoding demo_conf.txt in CyberChef

The data doesn't decode to anything that is human readable – maybe it's not actually Base64-encoded or there could be additional layers of encoding. We won't be able to answer this until we have completed more analysis.

Let's take a closer look at the flake.exe file. We run `strings.exe` and notice surprisingly few strings for such a large file – this is a good indicator that the file is compressed, packed, or otherwise obfuscated. We notice the following group of interesting strings:

```
Error, couldn't runtime expand target path.
Error, couldn't decode attached data.
Error, could find attached data header.
Error, couldn't allocate memory.
Error, failed to open '%ls' for writing.
Error, couldn't runtime expand spec '%ls'.
NUITKA_ONEFILE_PARENT
%TEMP%\onefile_%PID%_%TIME%
Error, failed to register signal handler.
Error, couldn't launch child
```

Specifically, there appears to be a file or directory path, `%TEMP%\onefile_%PID%_%TIME%`, and multiple error messages describing decompression and execution of a child process.

Opening the file `flake.exe` in CFF Explorer and Detect It Easy shows us that the file has a large resource section containing high entropy data. This is likely the compressed data mentioned in the file's strings:

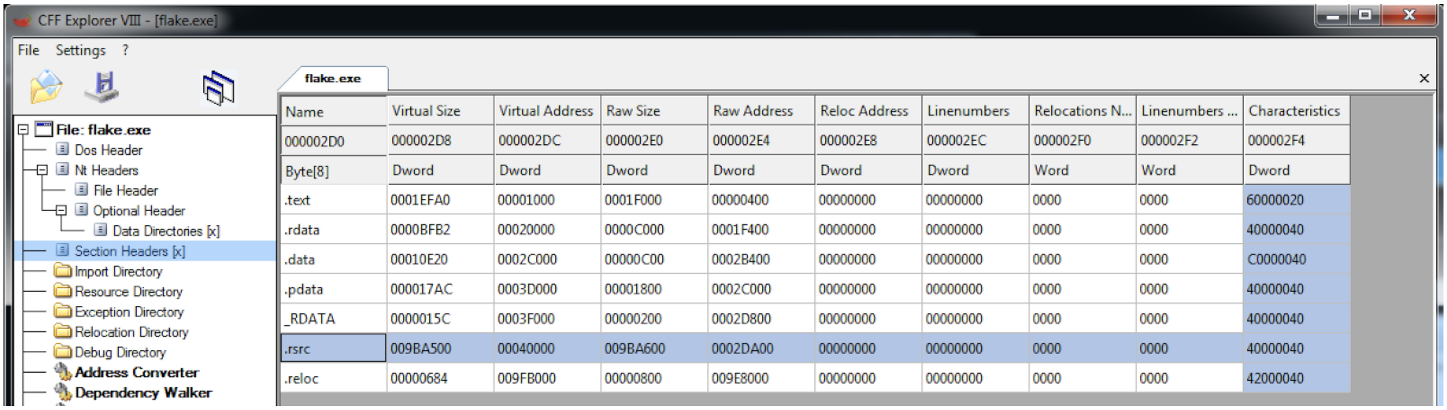

Figure 2: Viewing flake.exe in CFF Explorer

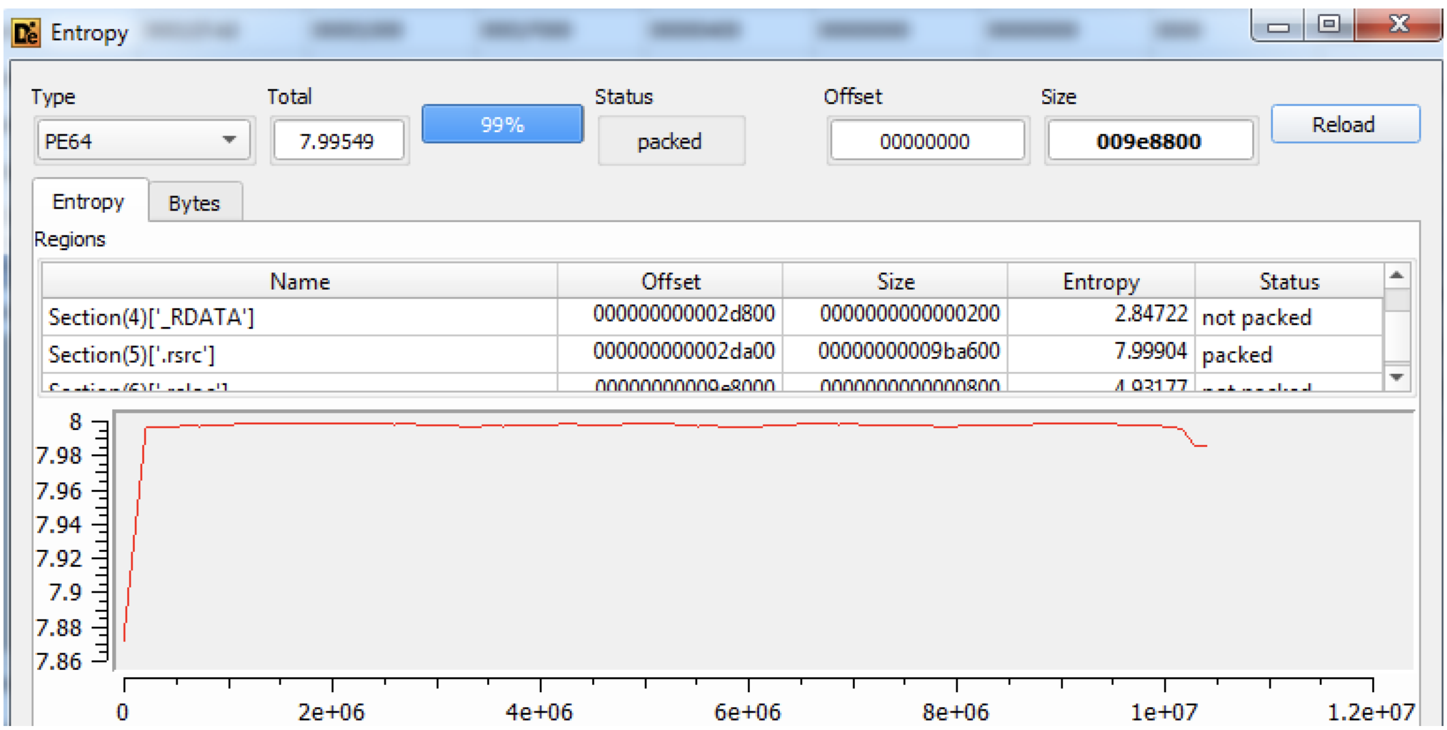

Figure 3: Viewing flake.exe in Detect It Easy

Based on these observations, we infer that the file `flake.exe` contains compressed data and, when executed, decompresses this data to the file or directory path `%TEMP%\onefile_%PID%_%TIME%` and executes it. Let's see if basic dynamic analysis can confirm.

Basic Dynamic Analysis: `flake.exe`

We open Process Monitor, set filters for `CreateFile`, `WriteFile`, and Process Create events generated by any process named `flake.exe`, and then execute the file `flake.exe` from the console:

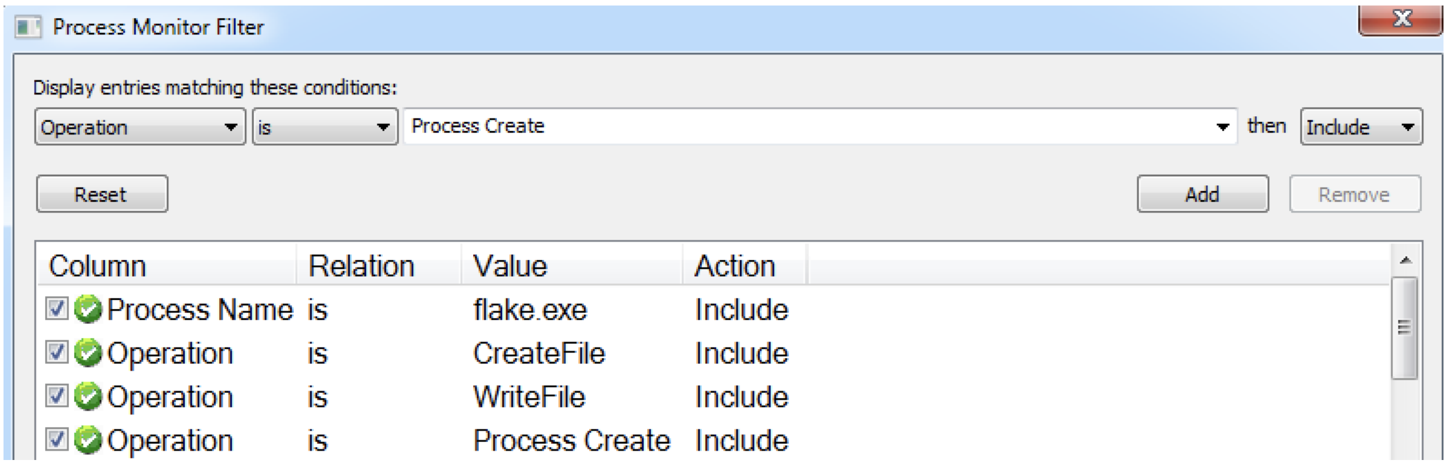

Figure 4: Setting Process Monitor filters

A new window opens that displays the game:

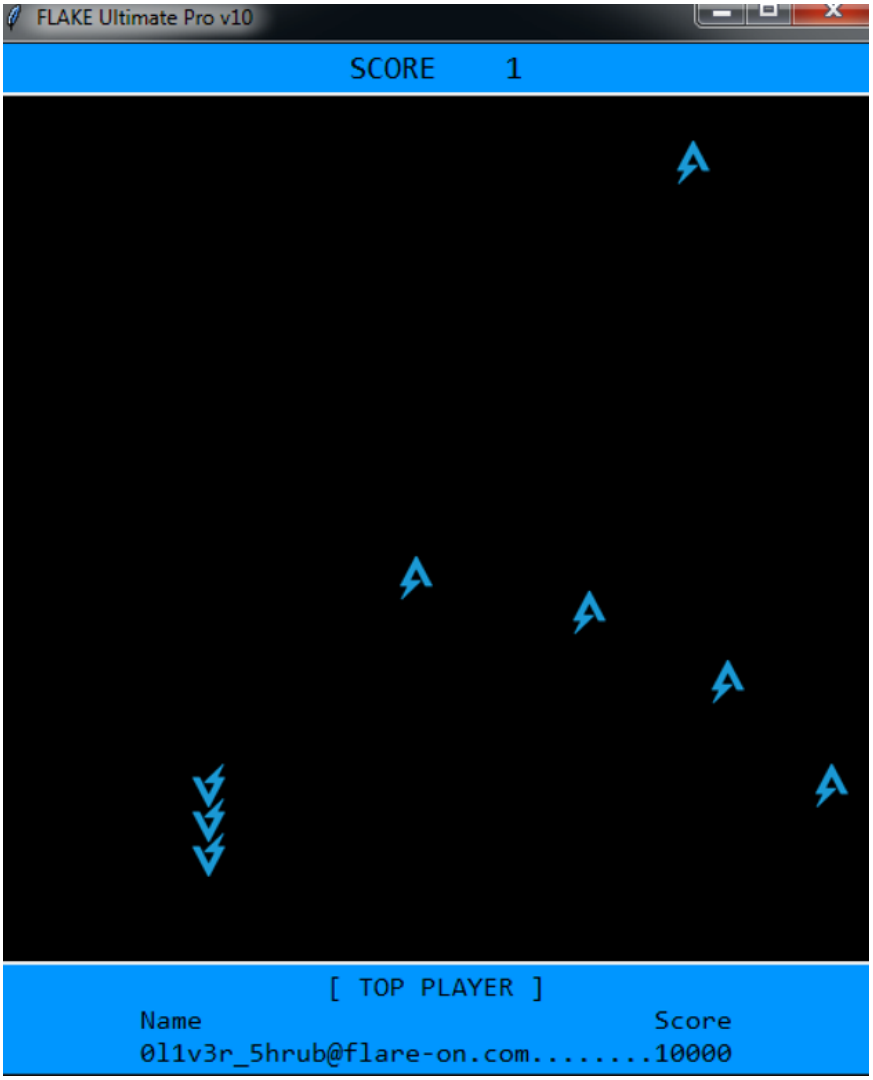

Figure 5: Viewing Flake

After some trial and error, we determine that the snake can be controlled using our keyboard arrows. As the snake consumes FLARE logos, we get points, and the snake moves faster until it's impossible to control:

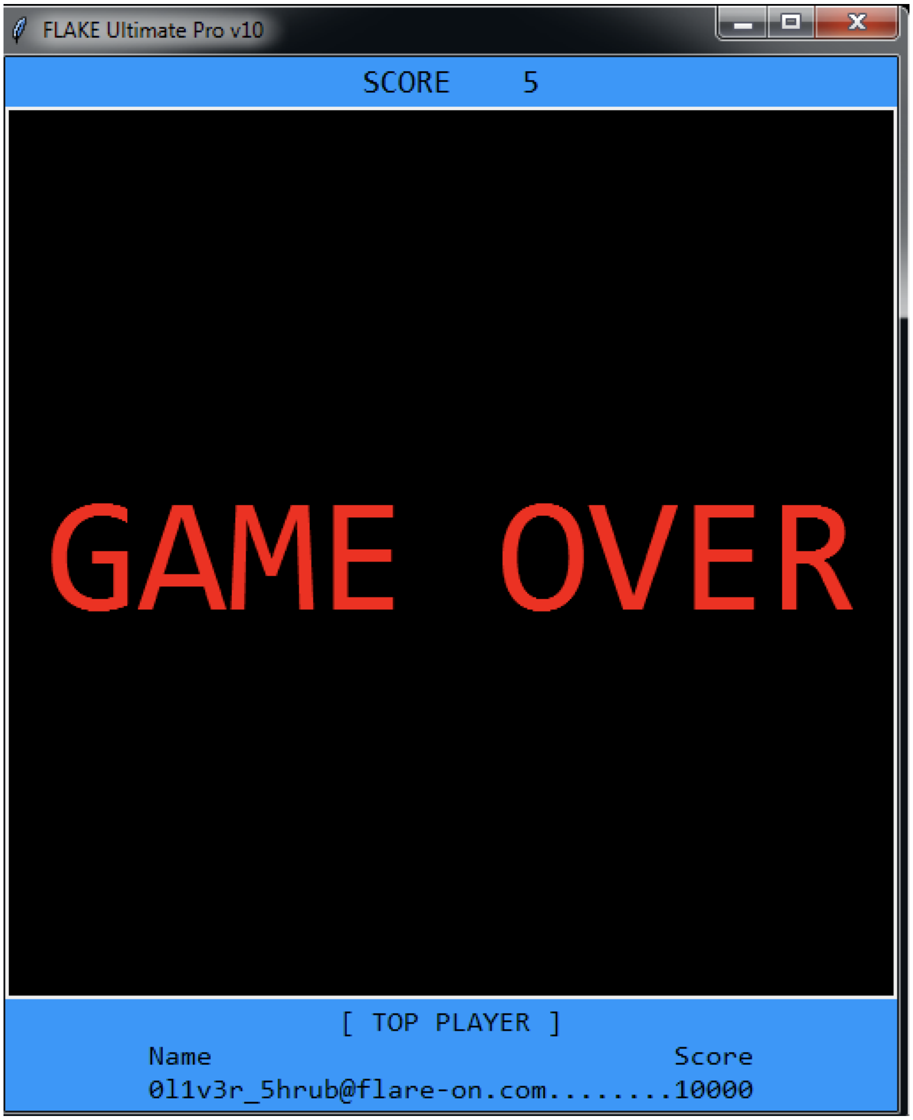

Figure 6: Viewing Flake "GAME OVER"

It looks like Nox is right that it is impossible to score 10,000 points without cheating. So how can we beat Oliver's high score? We see the following message printed to console when the file `flake.exe` is executed:

```
[!] could not find configuration file in directory . - using prod configuration
```

The program searches its current directory for a configuration file but does not see the file `demo_conf.txt`. Let's check Process Monitor.

Looking through the captured events, we see the program:

- Create the directory `C:\Users\user\AppData\Local\Temp\onefile_1792_133397202283992214`
- Write many files and subdirectories to the newly created directory

- Execute the file `C:\Users\user\AppData\Local\Temp\onefile_1792_133397202283992214\flake.exe` with process ID (PID) 1800

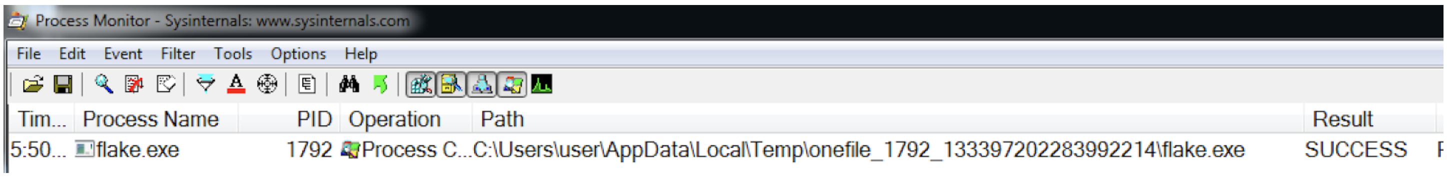

Figure 7: Viewing "Process Create" event in Process Monitor

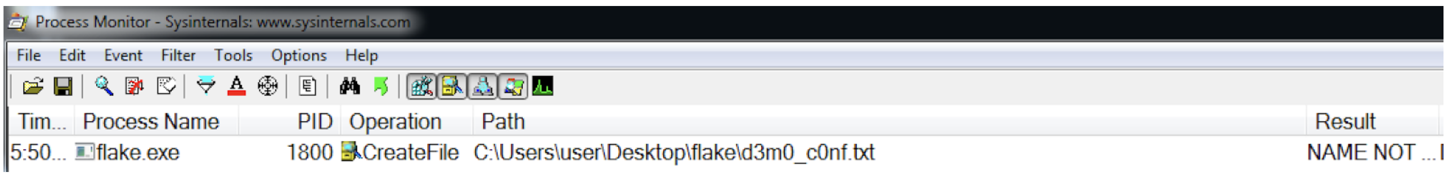

Figure 8: Viewing "CreateFile" event in Process Monitor

Let's refer to the file `C:\Users\user\AppData\Local\Temp\onefile_1792_133397202283992214\flake.exe` as `onefile_flake.exe` moving forward to help distinguish it from the original file `flake.exe`.

Process Monitor has confirmed what we learned from our basic static analysis. Next, we should analyze the file `onefile_flake.exe` as it appears to be the second stage of execution. But before we do that, let's see if we can learn anything about the missing "configuration file" mentioned earlier in the console output.

Identifying the Configuration File

Based on the message output to the console, the file `flake.exe` expects its configuration file to be stored in its working directory. We filter Process Monitor for `CreateFile` events generated by the process `flake.exe` in its working directory and see that the expected filename is `d3m0_c0nf.txt`:

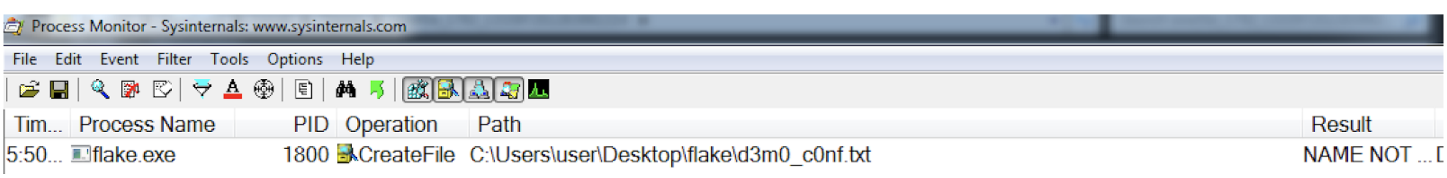

Figure 9: Viewing "CreateFile" events in Process Monitor

This filename is suspiciously like the existing filename `demo_conf.txt`. We also see that the process identifier that is linked to the `CreateFile` event is that of the file `onefile_flake.exe`, not the file `flake.exe`.

We change the filename from demo_conf.txt to d3m0_c0nf.txt, execute the file flake.exe, and the following message appears in the console:

```
[!] configuration file found and decoded with key - using demo configuration
```

We also notice that our score increases by two points instead of one indicating that the configuration file alters game mechanics. Maybe this is the key to beating Oliver's high score? We still don't know the format of the data stored in the file d3m0_c0nf.txt but we do know that the file onefile_flake.exe is responsible for reading and decoding it. Let's analyze the file onefile_flake.exe.

Basic Static Analysis: onefile_flake.exe

We run strings.exe on the file onefile_flake.exe and see many strings. We search for the string d3m0_c0nf.txt and find an interesting grouping of strings:

```
u[!] bad configuration file - using prod configuration
u[!] configuration file found and decoded with key - using demo configuration
nnnu[!] could not find configuration file in directory
u - using prod configuration
uXOR-encode d3m0_c0nf.txt with 0x22,0x11,0x91,0xff (I think Nuikta strips Python docstrings during compilation
so no worries about this comment making its way into the wrong hands)
```

The last string is very interesting:

```
uXOR-encode d3m0_c0nf.txt with 0x22,0x11,0x91,0xff (I think Nuikta strips Python docstrings during compilation
so no worries about this comment making its way into the wrong hands)
```

This string reveals two pieces of important information:

- onefile_flake.exe contains Python code that has been compiled using Nuitka
- d3m0_c0nf.txt is encoded using the multi-byte XOR key \x22\x11\x91\xff

Nuitka is an open source, optimizing Python compiler that "translates Python modules into a C level program that then uses libpython and static C files of its own to execute in the same way as CPython does". Reading Nuitka's user manual we learn that Nuitka-compiled Python projects can be distributed using standalone or onefile modes. Nutika onefile mode creates a single binary that extracts itself and all dependencies on the target, before running the target program. The file flake.exe is compiled using Nuitka onefile mode based on the strings, e.g. "NUITKA_ONEFILE_PARENT", and the behavior that we captured in Process Monitor.

We confirm that we can execute the file onefile_flake.exe directly from its directory C:\Users\user\AppData\Local\Temp\onefile_1792_133397202283992214. We copy the file d3m0_c0nf.txt to this directory so that the file onefile_flake.exe can find it. Let's figure out how we can decode the file d3m0_c0nf.txt.

Decoding d3m0_c0nf.txt

The file d3m0_c0nf.txt contains the following:

WTOh3Rgz17NjWtTfd33llk9w5ZoCQeOQAmegzwI51ZpPfrjdDjOg3Rgkvd0QM6vPDjOi3Rgj7A==

Earlier, we tried Base64 decoding the data using CyberChef, but the result was not human readable. We learned from the leaked Python docstring that there is an additional XOR encoding layer that uses the multi-byte XOR key \x22\x11\x91\xff.
Let's recreate this in CyberChef:

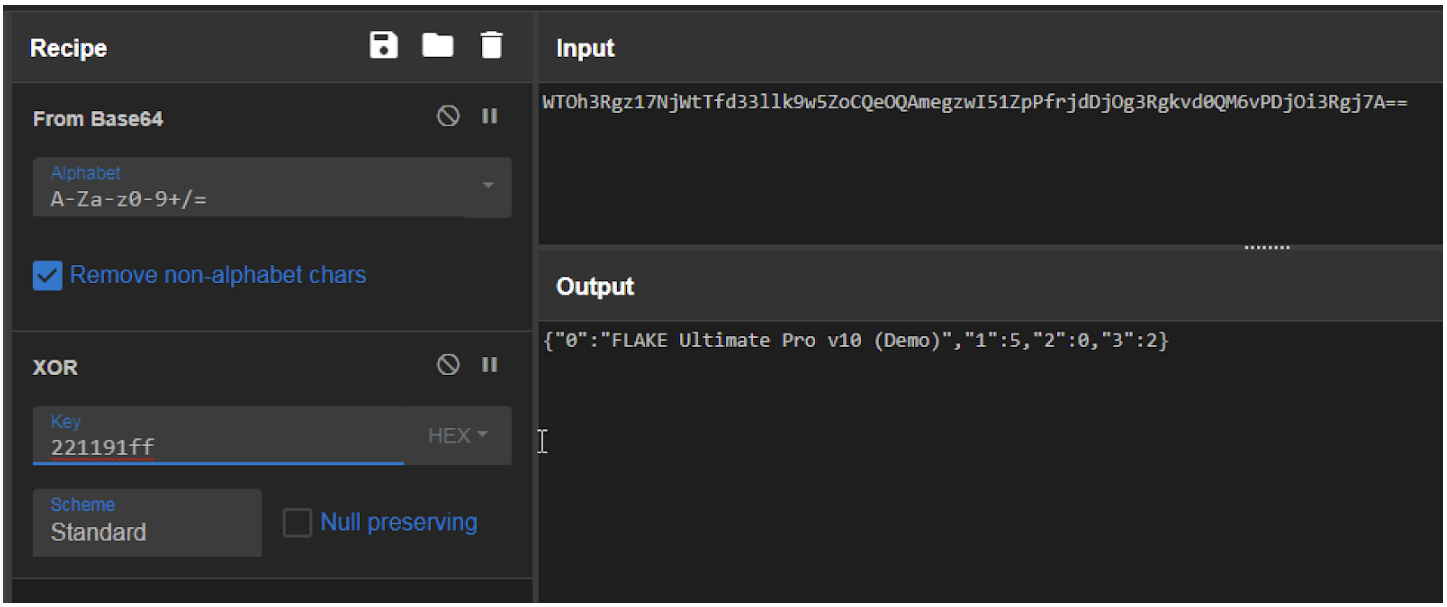

Figure 10: Decoding d3mo_c0nf.txt using CyberChef

Success! The file `d3m0_c0nf.txt` decodes to the JSON string:

{"0":"FLAKE Ultimate Pro v10 (Demo)","1":5,"2":0,"3":2}

Unfortunately, the JSON does not have human-readable key names, but we can guess some of their purposes. "0" is likely the game title and "3" is likely the point increase delta, based on our earlier observation that our score increased by two.

We use CyberChef to change the point increase delta, "3", from two to 10,000:

Figure 11: Modifying and encoding configuration using CyberChef

Encoding the modified JSON results in the following Base64-encoded string:

WTOh3Rgz17NjWtTfd33llk9w5ZoCQeOQAmegzwI51ZpPfrjdDjOg3Rgkvd0QM6vPDjOi3Rggoc8SIew=

We store this string in the file `d3m0_c0nf.txt` and execute the file `onefile_flake.exe`. Immediately we see that our score increases by 10,000 points instead of two, overtaking Oliver's high score!

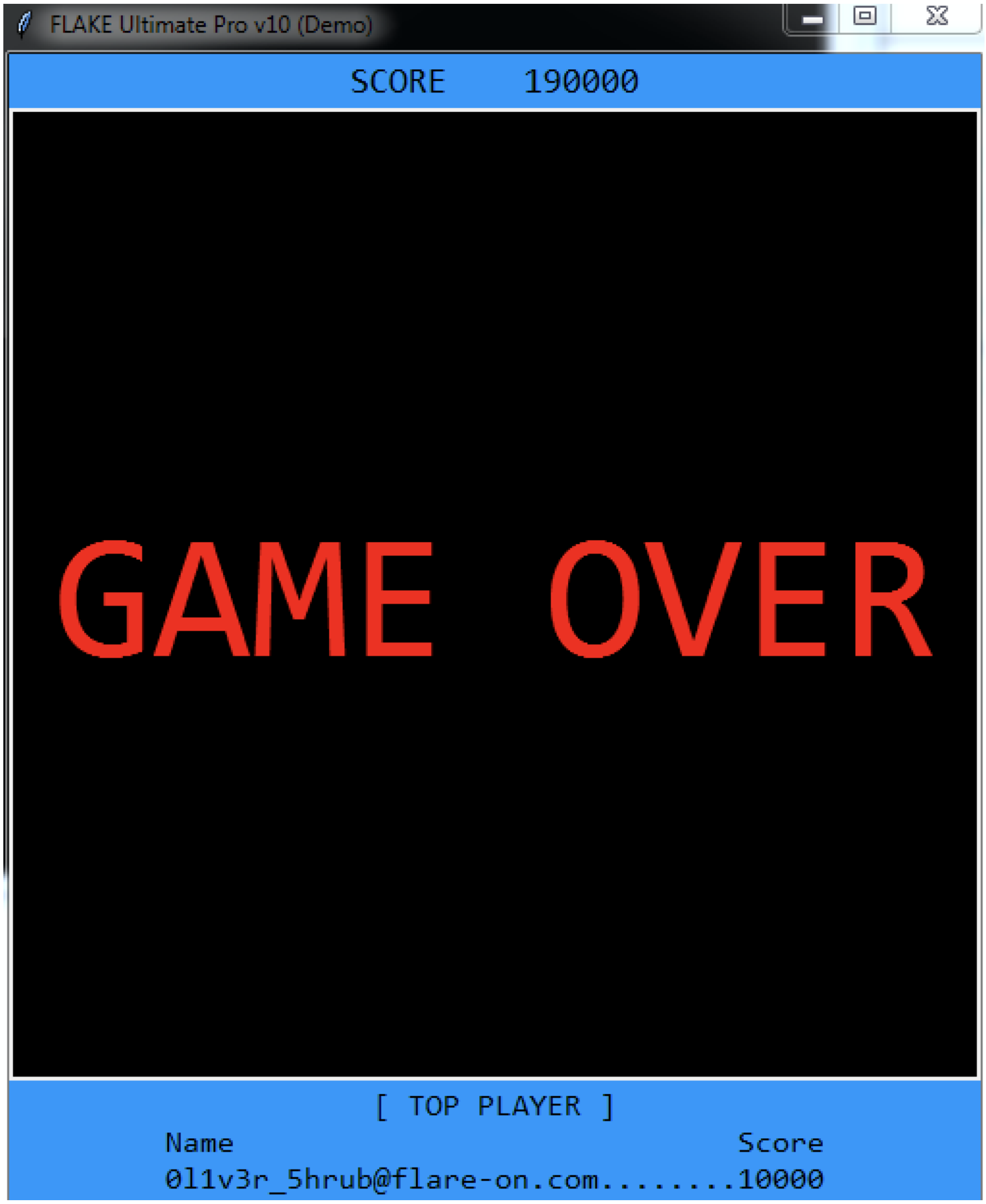

Figure 12: Viewing Flake large point increase

However, we still see “GAME OVER” and the following message is printed to the console:

```
[!] Snake.length property, not including start length, is 19 but it must equal the final score which is 190000!
```

This appears to be the result of an anti-cheat mechanism that is triggered if the game ends with a final score that does not match the snake’s length property. Experimenting with the other configuration values does not identify a key that affects the snake’s length.

Investigating the Anti-Cheat Mechanism

We search the strings output of the file `onefile_flake.exe` for strings containing "`snake`" and see the following groups of interesting strings:

```
uSnake.__init__
uSnake.update
uSnake.shift
uSnake.dump_index
aproperty
uSnake.length
uSnake.head
[...]
aget_flag
acheck_snake_length
ashame
uflag.py
u<module flag>
asnake
aexpected_length
```

The first group of strings appear related to a Python object named Snake. This is based on the first string, "`Snake.__init__`", where the Python keyword `__init__` identifies an object's constructor method. The other strings indicate additional methods or properties that the Snake object may implement, including a length property as mentioned in the console message.

The second group of strings appear related to checking the snake length, specifically the string "`check_snake_length`". Let's see if we can pivot into Ghidra using the string "`check_snake_length`". We open Ghidra, load and analyze `onefile_flake.exe`, and use Ghidra's "Define Strings" window to check for references to the string "`check_snake_length`". We didn't find any.

Let's see where the string is stored. We open Search > For Strings..., run a new search, and filter for the string "`check_snake_length`". This identifies that the string is stored in `onefile_flake.exe`'s resource section:

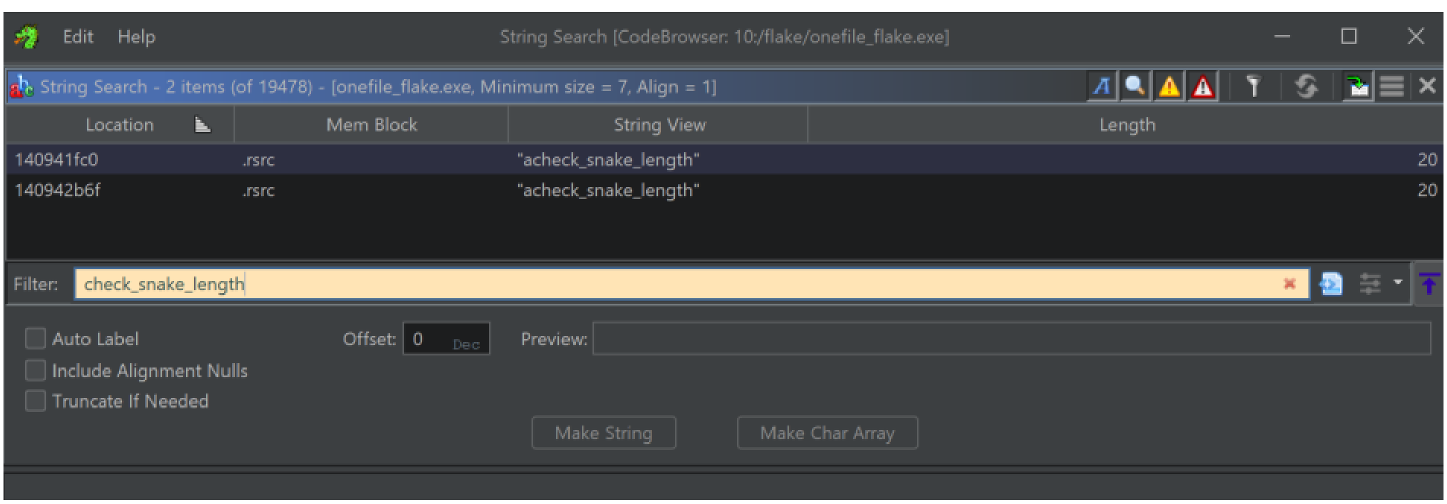

Figure 13: Searching for "check_snake_length" in Ghidra

We identify that all the strings shown earlier are stored in `onefile_flake.exe`'s resource section. Using Ghidra's "Symbol Tree" window, we search for imports related to resource manipulation:

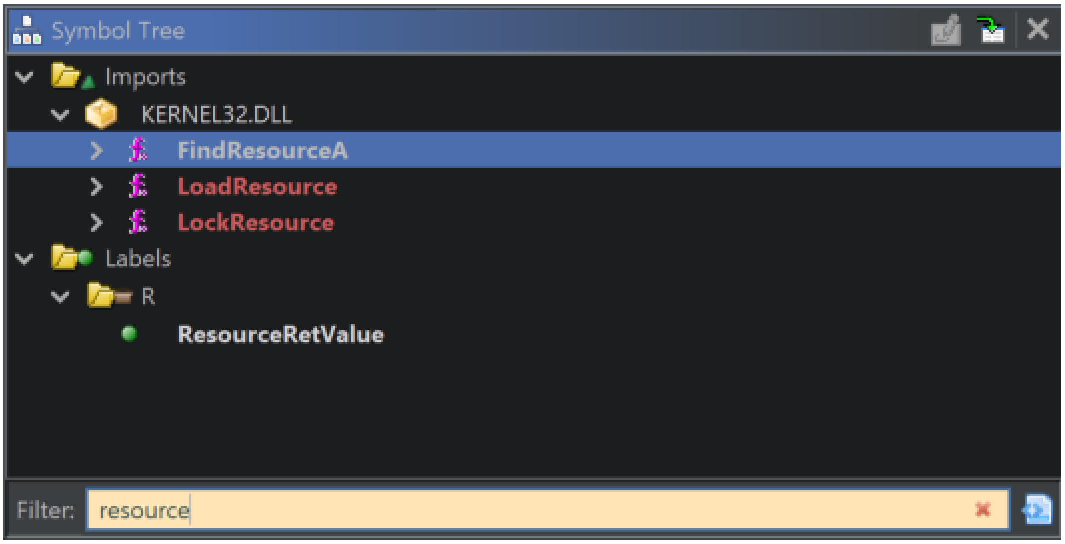

Figure 14: Searching for imports in Ghidra

We pivot on the import `FindResourceA` and identify the function FUN_1404BC0E0 that is responsible for reading and unpacking `onefile_flake.exe`'s resource section:

```c

void FUN_1404bc0e0(undefined8 param_1,undefined8 param_2,byte *param_3)

{
  byte bVar1;
  uint uVar2;
  int iVar3;
  int iVar4;
  HRSRC hResInfo;
  HGLOBAL hResData;
  int *piVar5;
  undefined8 uVar6;
  byte *pbVar7;
  longlong lVar8;
  ulonglong uVar9;
  longlong lVar10;
  uint uVar11;
  undefined8 *puVar12;
  int *piVar13;
  undefined2 *puVar14;

  piVar13 = DAT_14052f970;
  if (DAT_14052fbbf == '\0') {
    hResInfo = FindResourceA((HMODULE)0x0,(LPCSTR)0x3,(LPCSTR)0xa);
    hResData = LoadResource((HMODULE)0x0,hResInfo);
    piVar5 = (int *)LockResource(hResData);
    iVar4 = *piVar5;
    piVar13 = piVar5 + 2;
    DAT_14052f970 = piVar13;
    iVar3 = FUN_1404b9d60(piVar13,piVar5[1]);
    if (iVar3 != iVar4) {
      FUN_1404ce710("Error, corrupted constants object");
                    /* WARNING: Subroutine does not return */
      abort();
    }
    DAT_14052fbbf = '\x01';
  }
  iVar4 = strcmp((char *)param_3,".bytecode");
```

Figure 15: Viewing FUN_1404BC0E0 in Ghidra

The data stored in `onefile_flake.exe`'s resource section appears to store constants used during execution, based on the string "`Error, corrupted constants object`" identified in the function `FUN_1404BC0E0`. Let's figure out how these constants are unpacked and used by the program so that we can identify how the string "`check_snake_length`" is referenced.

Advanced Static Analysis: `onefile_flake.exe`

We rename the function `FUN_1404BC0E0` to `zUnpackConstants`. Taking a closer look at the function `zUnpackConstants` we identify that the first eight bytes store a 4-byte CRC32 hash followed by a 4-byte size of the constants object:

```
x_const_obj = g_const_obj;
if (g_const_obj_init == '\0') {
  hResInfo = FindResourceA((HMODULE)0x0,(LPCSTR)0x3,(LPCSTR)0xa);
  hResData = LoadResource((HMODULE)0x0,hResInfo);
  x_const_res = (BYTE *)LockResource(hResData);
  x_crc32_hash = *(UINT32 *)x_const_res;
  x_const_obj = x_const_res + 8;
  g_const_obj = x_const_obj;
  x_calc_crc32_hash = zCalcCrc32Hash(x_const_obj,*(UINT32 *)(x_const_res + 4));
  if (x_calc_crc32_hash != x_crc32_hash) {
    zDoError("Error, corrupted constants object");
                    /* WARNING: Subroutine does not return */
    abort();
  }
  g_const_obj_init = '\x01';
}
```

Figure 16: Viewing constants object hash verification in Ghidra

We determine that the hashing used is CRC32 based on the constant 0xEDB88320 that we identify in the function FUN_1404B9D60:

```
UINT32 zCalcCrc32Hash(BYTE *data,UINT32 size)

{
  byte bVar1;
  uint uVar2;
  ulonglong uVar3;

  uVar2 = 0xffffffff;
  if (size != 0) {
    uVar3 = (ulonglong)size;
    do {
      bVar1 = *data;
      data = data + 1;
      uVar2 = -(uint)(((uVar2 ^ bVar1) & 1) != 0) & 0xedb88320 ^ (uVar2 ^ bVar1) >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar2 = -(uint)((uVar2 & 1) != 0) & 0xedb88320 ^ uVar2 >> 1;
      uVar3 = uVar3 - 1;
    } while (uVar3 != 0);
  }
  return ~uVar2;
}
```

Figure 17: Viewing FUN_1404B9D60 in Ghidra

Further analysis of `zUnpackConstants` reveals that following the 8-byte header is one or more constants blobs. Each blob starts with a variable length ASCII string followed by a 4-byte size of the blob followed by a 2-byte unknown value followed by the blob data. The function `zUnpackConstants` accepts an ASCII string as its third argument that it compares to each constants blob name until a match is found. If a match is found, the function passes a pointer to the matched constants blob and the unknown 2-byte value to the function `FUN_1404B8FB0` at the address 0x1404BC2F1:

```
x_temp_obj_size = *(uint *)(x_const_obj + 1 + (longlong)x_temp_obj_name_end);
x_temp_const_blob = x_const_obj + (longlong)x_temp_obj_name_end + 5;
do {
  if (x_strmp_result == 0) {
    FUN_1404b8fb0(param_1,param_2,(BYTE *)((longlong)x_temp_const_blob + 2),
                  (uint)*(ushort *)x_temp_const_blob);
    return;
  }
```

Figure 18: zUnpackConstants passing constants blob pointer and unknown 2-byte value to FUN_1404B8FB0

We analyze the function `FUN_1404B8FB0` and determine that it is responsible for unpacking each constant into a specified Python type. The unknown 2-byte value stores the number of constants to unpack. We rename the function `FUN_1404B8FB0` to `zUnpackConstantsBlob`. Each constant stored in a blob starts with a single byte that identifies the constant's Python type shown at the address 0x1404B8FFC:

```
if (0 < p_const_count) {
  do {
    puVar18 = puVar17;
    x_pyobj_list_curr = local_c8;
    *local_c8 = (void *)0x0;
    x_pyobj_type = *p_const_blob;
    x_const_blob_index_curr = (BYTE *)((longlong)p_const_blob + 1);
    pcVar11 = (char *)x_const_blob_index_curr;
    puVar17 = puVar18;
    puVar20 = (undefined8 *)x_const_blob_index_curr;
    switch(x_pyobj_type) {
```

Figure 19: Viewing constant type identifier usage at 0x1404B8FFC in Ghidra

For example, each of the following strings starts with an “a” or “u”:

```
aget_flag
acheck_snake_length
ashame
uflag.py
u<module flag>
asnake
aexpected_length
```

We identify the code in `zUnpackConstantsBlob` that handles unpacking the “a” and “u” Python types and find that the corresponding constants are unpacked into Python Unicode objects via the function [PyUnicode_DecodeUTF8](#) shown at the address 0x1404B96D9:

```
case 'a':
case 'u':
  x_const_size = -1;
  do {
    lVar16 = x_const_size;
    x_const_size = lVar16 + 1;
  } while (x_const_blob_index_curr[x_const_size] != '\0');
  *(undefined8 *)(puVar18 + -8) = 0x1404b96fc;
  x_pyobj = (void *)PyUnicode_DecodeUTF8(x_const_blob_index_curr,x_const_size,"surrogatepass")
  ;
  x_const_blob_index_curr = (BYTE *)((longlong)p_const_blob + lVar16 + 3);
  if (x_pyobj_type == 'a') {
    *(undefined8 *)(puVar18 + -8) = 0x1404b9716;
    PyUnicode_InternInPlace(&x_pyobj);
  }
  *x_pyobj_list_curr = x_pyobj;
  p_const_count = x_const_count;
  break;
```

Figure 20: Viewing "a" and "u" unpacking at 0x1404B96D9 in Ghidra

The function `zUnpackConstantsBlob` stores each newly created Python object into an array that is passed as the function’s second argument:

```
void zUnpackConstantsBlob
               (void *param_1,void **p_pyobject_array,BYTE *p_const_blob,INT32 p_const_count)
```

Figure 21: Viewing zUnpackConstantsBlob parameters in Ghidra

This array is passed to the function `zUnpackConstantsBlob` from the function `zUnpackConstants`. The function `zUnpackConstants` has many cross references. We follow the cross reference from the address 0x140004DE5 and see that the array used to store Python objects belonging to the constants blob named “`Crypto.Cipher.ARC4`” is located in `onefile_flake.exe`’s data section at the address 0x14050A550:

```
zUnpackConstants(param_1,&DAT_14050a550,"Crypto.Cipher.ARC4");
```

Figure 22: Viewing call to zUnpackConstants at 0x140004DE5 in Ghidra

We see that the Python objects stored in this array are referenced by other code in the program. Therefore, by parsing the constants blobs we can determine where the resulting Python objects are stored and referenced. This is exactly what we need to help us identify where the string "`check_snake_length`" is used by the program.

Parsing the Constants Blobs

Using what we learned from our analysis of the functions `zUnpackConstants` and `zUnpackConstantsBlob`, let's write a Python script to unpack the constants blobs. First, we write `onefile_flake.exe`'s resource section to a file using CFF Explorer's "Resource Editor":

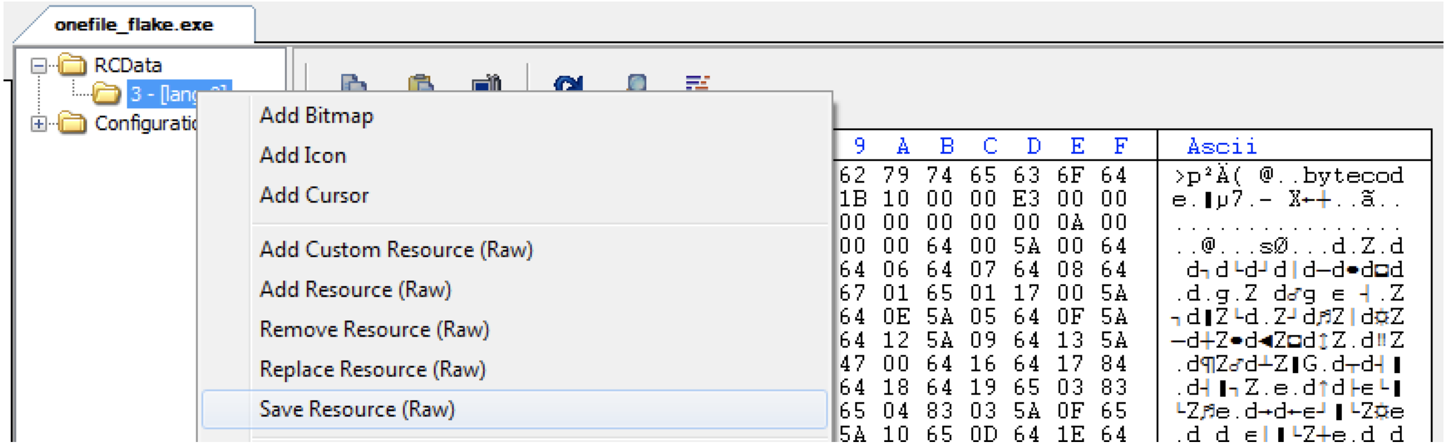

Figure 23: Writing onefile_flake.exe resource section to file using CFF Explorer

The first iteration of our Python script prints the name, size, and number of constants for each constants blob to help us better understand what is being stored:

```
import io
import sys
import struct


def read_uint32(bio):
    return struct.unpack("<I", bio.read(4))[0]


def read_uint16(bio):
    return struct.unpack("<H", bio.read(2))[0]


def read_utf8(bio):
    bs = b""
    while True:
        bs += bio.read(1)
        if b"\x00" in bs:
            break
    return bs[:-1].decode("utf-8")


def main():
    with open(sys.argv[1], "rb") as f_in:
        bs = f_in.read()

    bio = io.BytesIO(bs)
```

```
    hash_ = read_uint32(bio)
    size = read_uint32(bio)

    print(f"hash: {hex(hash_)}")
    print(f"size: {hex(size)}")

    while bio.tell() < size:
        blob_name = read_utf8(bio)
        blob_size = read_uint32(bio)
        blob_count = read_uint16(bio)

        print(f"name: {blob_name}, size: {hex(blob_size)}, count: {hex(blob_count)}")

        bio.seek(bio.tell() + (blob_size - 2))


if __name__ == "__main__":
    main()
```

Which produces the following output:

```
hash: 0x5b855ecc
size: 0x3fd88a
name: .bytecode, size: 0x377242, count: 0x12a
name: , size: 0x2e5, count: 0x57
name: Crypto.Cipher.ARC4, size: 0xaa7, count: 0x3c
name: Crypto.Cipher._EKSBlowfish, size: 0xb4f, count: 0x35
name: Crypto.Cipher._mode_cbc, size: 0x17f9, count: 0x5b
name: Crypto.Cipher._mode_ccm, size: 0x32de, count: 0xb9
name: Crypto.Cipher._mode_cfb, size: 0x19a3, count: 0x5d
name: Crypto.Cipher._mode_ctr, size: 0x2420, count: 0x78
[...]
name: PIL._version, size: 0x81, count: 0xb
name: PIL, size: 0x75f, count: 0x24
name: PIL.features, size: 0x1531, count: 0x86
name: __main__, size: 0xecb, count: 0x101
name: flag, size: 0x1b8, count: 0x24
name: tkinter-preLoad, size: 0xbc, count: 0x11
```

Each blob name appears to correspond to a Python module e.g. the constants blob named "`Crypto.Cipher.ARC4`" likely stores constants related to PyCryptodome's ARC4 module. Additionally, the following module names stand out:

- `__main__`
- `flag`

Python's `__main__` is the first user-specified Python module that is executed. We don't know what the flag module does, but earlier we identified strings related to "getting" a flag that were stored near the strings related to checking the snake length. Let's extend our Python script to write the `__main__` and flag constants blobs to files named `__main__.bin` and `flag.bin`, respectively:

```
[...]
    while bio.tell() < size:
        blob_name = read_utf8(bio)
        blob_size = read_uint32(bio)
        blob_count = read_uint16(bio)
```

```
        print(f"name: {blob_name}, size: {hex(blob_size)}, count: {hex(blob_count)}")

        if blob_name == "__main__" or blob_name == "flag":
            with open(f"{blob_name}.bin", "wb") as f_out:
                f_out.write(bio.read(blob_size - 2))
        else:
            bio.seek(bio.tell() + (blob_size - 2))
[...]
```

We run strings.exe on the file `__main__.bin` and see many strings related to core game logic, including the configuration file related strings that we analyzed earlier:

```
u[!] bad configuration file - using prod configuration
u[!] configuration file found and decoded with key - using demo configuration
nnnu[!] could not find configuration file in directory
u - using prod configuration
uXOR-encode d3m0_c0nf.txt with 0x22,0x11,0x91,0xff (I think Nuikta strips Python docstrings during compilation
so no worries about this comment making its way into the wrong hands)
```

We run `strings.exe` on the file `flag.bin` and see, among others, strings related to checking the snake's length, including our target string "`check_snake_length`":

```
aARC4
anew
adecrypt
adecode
uutf-8
alength
aprint
uCrypto.Cipher
aARC4
aget_flag
acheck_snake_length
ashame
uflag.py
u<module flag>
asnake
aexpected_length
aactual
aexpected
```

Let's expand our Python script to unpack the flag module's constants:

```
import io
import sys
import struct


def read_uint8(bio):
    return struct.unpack("<B", bio.read(1))[0]


def read_uint16(bio):
    return struct.unpack("<H", bio.read(2))[0]


def read_uint32(bio):
    return struct.unpack("<I", bio.read(4))[0]


def read_utf8_size_1(bio):
```

```
    return bio.read(1).decode("utf-8")


def read_utf8(bio):
    bs = b""
    while True:
        bs += bio.read(1)
        if b"\x00" in bs:
            break
    return bs[:-1].decode("utf-8")


def read_bytearray(bio):
    bs = b""
    while True:
        bs += bio.read(1)
        if b"\x00" in bs:
            break
    return bs[:-1]

def decode_blob(bio, count):

    container = []

    for i in range(count):
        type_ = chr(read_uint8(bio))

        if type_ in ('a', 'u'):
            o = read_utf8(bio)
        elif type_ == 'l':
            o = read_uint32(bio)
        elif type_ == 'w':
            o = read_utf8_size_1(bio)
        elif type_ == 'T':
            sub_count = read_uint32(bio)
            o = tuple(decode_blob(bio, sub_count))
        elif type_ == 'c':
            o = read_bytearray(bio)
        elif type_ == 'b':
            size = read_uint32(bio)
            o = bio.read(size)
        else:
            raise ValueError(f"unhandled type {type_}")

        container.append(o)

    return container


def main():
    with open(sys.argv[1], "rb") as f_in:
        bs = f_in.read()

    bio = io.BytesIO(bs)

    hash_ = read_uint32(bio)
    size = read_uint32(bio)

    #print(f"hash: {hex(hash_)}")
    #print(f"size: {hex(size)}")

    while bio.tell() < size:
        blob_name = read_utf8(bio)
        blob_size = read_uint32(bio)
        blob_count = read_uint16(bio)

        #print(f"name: {blob_name}, size: {hex(blob_size)}, count: {hex(blob_count)}")
```

```
        if blob_name == "flag":
            decoded = decode_blob(bio, blob_count)
            for idx, o in enumerate(decoded):
                print(f"{idx}: {o}")
        else:
            bio.seek(bio.tell() + (blob_size - 2))


if __name__ == "__main__":
    main()
```

The big addition here is the Python function `decode_blob`. Starting with the types “a” and “u” we implement unpacking code for each constant type used by the flag module. This includes:

- “a” and “u”: unpack Python Unicode object (see address 0x1404B96D9)
- “l”: unpack Python Integer object (see address 0x1404B93A6)
- “w”: unpack Python Unicode object (see address 0x1404B965F)
- “T”: unpack Python Tuple object containing one or more Python objects (see address 0x1404B9054)
- “c”: unpack Python Bytes object (see address 0x1404B9605)
- “b”: unpack Python Bytes object (see address 0x1404B968C)

Which results in the following output:

```
0: 0
1: dk
2: append
3: i
4: 1
5: ARC4
6: new
7: decrypt
8: (b"\xbbh\xd5P\x88\xc3$\x1bM\xdc\xc2\x9d\x89\xaafGx\xa6\xdb\x82\x02\xc6V\xce\xbb\x95@\x7f'*`\xee\xc0i",)
9: decode
10: ('utf-8',)
11: length
12:
b'Z#^$Rlbod,oaoewl!rqkqgqpx.#jnv#moaoqekmc!qwesv#hdldpi.#mr"&`!`vp!kw$lwpp!grq`n#pig#bhlbh!q`ksg#sik`l!kp$$f"'
13: dm
14: b'\x01\x02\x03\x04'
15: 4
16: print
17: __doc__
18: __file__
19: __spec__
20: origin
21: has_location
22: __cached__
23: Crypto.Cipher
24: ('ARC4',)
25:
(b'T\x00\xc6\x88g\xf9_nx}\x91]X\xb2^g[\xf40\x860\xe4D\x19\xea\x94\x136\x97m\xc9\xd8\xb9r?(\xe8\xea\r3\x92\x8e\xa
9\x03\xef\xa8\x8e\x9d\xb7\x83',)
26: get_flag
27: check_snake_length
28: shame
29: flag.py
30: <module flag>
31: ('snake', 'expected_length')
32: ('xk', 'k', 'c', 'dk', 'i', 'b', 'p')
```

```
33: ('actual', 'expected', 'xk', 'em', 'dm', 'i', 'b')
34:
```

We now know the index and value of each constant used by the flag module, including our target string "`check_snake_length`". We also know that the function `zUnpackConstants` accepts parameters including the name of the target constants blob and the address where the resulting Python objects are stored. Let's see if we can identify where the flag module's Python objects are stored and referenced.

Analyzing the `flag` Module's Python Objects

We search for the string "`flag`" in Ghidra:

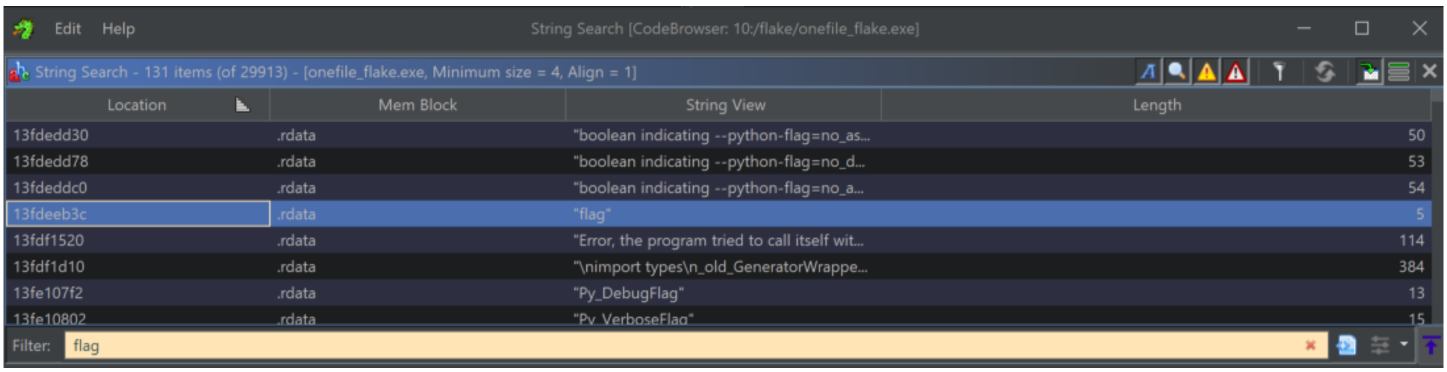

Figure 24: Searching for "flag" in Ghidra

By analyzing the cross references to the string "`flag`" we identify a call to the function `zUnpackConstants` at the address 0x14048840E that is responsible for unpacking the flag module's constants:

```
zUnpackConstants(param_1,&DAT_14052f740,"flag");
```

Figure 25: Viewing call to zUnpackConstants at 0x14048840E in Ghidra

We now know the address 0x14052F740 where the flag module's Python objects are stored. Using the results of our Python script we can set the data type at this address to an array of 35 pointers to help us identify how each Python object is referenced:

```
                    g_flag_module_constants
14052f740             void …   ??
  14052f740             void * NaP            [0]                     XREF[9]:  140486929(R),
                                                                                140486c3b(R),
                                                                                140486c6d(R),
                                                                                FUN_140487880:1404878d
                                                                                FUN_140487880:140487b5
                                                                                FUN_140487880:140487b8
                                                                                FUN_1404883d0:14048840
                                                                                FUN_1404883d0:14048882
                                                                                FUN_1404883d0:14048888
  14052f748             void * NaP            [1]                     XREF[2]:  140486ed0(R),
                                                                                1404871b1(R)
  14052f750             void * NaP            [2]                     XREF[2]:  140486b21(R),
                                                                                FUN_140487880:140487a5
  14052f758             void * NaP            [3]                     XREF[2]:  140486e38(R),
                                                                                FUN_140487880:140487d5
  14052f760             void * NaP            [4]                     XREF[2]:  140486bdc(R),
                                                                                FUN_140487880:140487b1
  14052f768             void * NaP            [5]                     XREF[5]:  14048712c(R),
                                                                                14048714c(R),
                                                                                FUN_1404883d0:14048887
                                                                                FUN_1404883d0:14048889
                                                                                FUN_1404883d0:14048898
  14052f770             void * NaP            [6]                     XREF[1]:  140487171(R)
  14052f778             void * NaP            [7]                     XREF[1]:  1404872d2(R)
  14052f780             void * NaP            [8]                     XREF[1]:  1404872c8(R)
  14052f788             void * NaP            [9]                     XREF[2]:  1404872f3(R),
                                                                                FUN_140487880:140487f8
  14052f790             void * NaP            [10]                    XREF[2]:  14048732e(R),
                                                                                FUN_140487880:140487fe
  14052f798             void * NaP            [11]                    XREF[1]:  14048767c(R)
  14052f7a0             void * NaP            [12]                    XREF[1]:  FUN_140487880:1404878d
  14052f7a8             void * NaP            [13]                    XREF[2]:  FUN_140487880:140487de
                                                                                FUN_140487880:140487ee
  14052f7b0             void * NaP            [14]                    XREF[1]:  FUN_140487880:140487a8
  14052f7b8             void * NaP            [15]                    XREF[1]:  FUN_140487880:140487b3
  14052f7c0             void * NaP            [16]                    XREF[1]:  FUN_140487880:140487eb
  14052f7c8             void * NaP            [17]                    XREF[1]:  FUN_1404883d0:1404886e
  14052f7d0             void * NaP            [18]                    XREF[1]:  FUN_1404883d0:14048870
  14052f7d8             void * NaP            [19]                    XREF[4]:  FUN_1404883d0:14048874
                                                                                FUN_1404883d0:14048876
                                                                                FUN_1404883d0:1404887a
                                                                                FUN_1404883d0:1404887d
```

Figure 26: Changing data type at 0x14052F740 to void*[35] in Ghidra

We see that our target string “check_snake_length” (index 27) and the Tuple ('snake', 'expected_length') (index 31) are passed as arguments to the function FUN_1404BCD10 at the address 0x14048849F.
Further inspection of the function FUN_1404BCD10 reveals that it creates a new Python Code object via the CPython function [PyCode_NewWithPosOnlyArgs] where the function name is check_snake_length and the parameter names are snake and expected_length. We rename FUN_1404BCD10 to zCreateCodeObject.
We see that the check_snake_length Code object is stored at the address 0x14052F870. We rename the address 0x14052F870 to g_check_snake_length_co:

```
g_check_snake_length_co =
     zCreateCodeObject(DAT_14052f858,0x70,0x43,g_flag_module_constants[27],
                       g_flag_module_constants[31],0,2,uVar17,uVar19);
```

Figure 27: Viewing "check_snake_length" Code object creation in Ghidra

By analyzing g_check_snake_length_co’s cross references we identify that it is passed as an argument to the function FUN_1404A2520 at the address 0x14048764C. Further inspection of the surrounding function, FUN_1404875F0, shows that the string “length” (index 11) is passed as an argument to the function FUN_1404A1DB0 at the address 0x140487688.
We analyze the function FUN_1404A1DB0 and see the string "'%s' object has no attribute ‘%s’". This indicates that the function FUN_1404A1DB0 may be used to retrieve a Python object’s attribute by name and we rename the function to zGetPythonObjectAttribute. We direct our analysis back to the function FUN_1404875F0.
We see that the Python object returned by the function zGetPythonObjectAttribute is passed to the function FUN_1404A7B10 at the address 0x14048769B. The value that is returned by the function FUN_1404A7B10 determines whether a True or False Python object is returned by the function FUN_1404875F0:

```
x_length_object =
     (longlong *)zGetPythonObjectAttribute(param_1,plVar4,g_flag_module_constants[11]);
if (x_length_object != (longlong *)0x0) {
  x_is_true_or_false = FUN_1404a7b10(x_length_object,plVar5);
  *x_length_object = *x_length_object + -1;
  if (*x_length_object == 0) {
    (**(code **)(x_length_object[1] + 0x30))(x_length_object);
  }
  if (x_is_true_or_false != -1) {
    x_retval = (longlong *)_Py_TrueStruct_exref;
    if (x_is_true_or_false != 1) {
      x_retval = (longlong *)_Py_FalseStruct_exref;
    }
```

Figure 28: Viewing FUN_1404875F0's return value in Ghidra

We now know that the function `FUN_1404875F0` returns a True or False Python object, references the `check_snake_length` Code object, and calls the function `zGetPythonObjectAttribute` to retrieve a Python object attribute named length. This appears to be the code that is responsible for verifying that the snake's length property equals our final score.
Let's see if we can identify the second argument that is passed to the function `FUN_1404A7B10` at the address 0x14048769B.
Starting at the address 0x140487607 we see that both the local variables `plVar5` and `plVar4` are initialized from an array of pointers passed as the third argument to the function `FUN_1404875F0`:

```
plVar4 = *param_3;
plVar5 = param_3[1];
```

Figure 29: Viewing plVar4 and plVar5 initialization in Ghidra

We know from our earlier analysis that the `check_snake_length` Code object is created with two parameters named `snake` and `expected_length`. We rename the local variables `plVar4` and `plVar5` to `x_snake_object` and `x_expected_length_object`, respectively, to confirm our suspicion that the function `FUN_1404875F0` is responsible for checking the snake's length:

```
x_length_object =
     (longlong *)zGetPythonObjectAttribute(param_1,x_snake_object,g_flag_module_constants[11]);
if (x_length_object != (longlong *)0x0) {
  x_is_true_or_false = zComparePythonObjects(x_length_object,x_expected_length);
  *x_length_object = *x_length_object + -1;
  if (*x_length_object == 0) {
    (**(code **)(x_length_object[1] + 0x30))(x_length_object);
  }
  if (x_is_true_or_false != -1) {
    x_retval = (longlong *)_Py_TrueStruct_exref;
    if (x_is_true_or_false != 1) {
      x_retval = (longlong *)_Py_FalseStruct_exref;
    }
```

Figure 30: Renaming plVar4 and plVar5 in Ghidra

Let's use x64dbg to determine if changing whether the function `FUN_1404875F0` returns a True or False Python object allows us to bypass the snake length verification.

Advanced Dynamic Analysis: `onefile_flake.exe`

We load the file `onefile_flake.exe` into x64dbg and set a breakpoint at the address 0x14048769B (0x13FD9769B after rebasing). We continue execution, play the game until the score displays 20,000, immediately lose, and see that our breakpoint is hit!

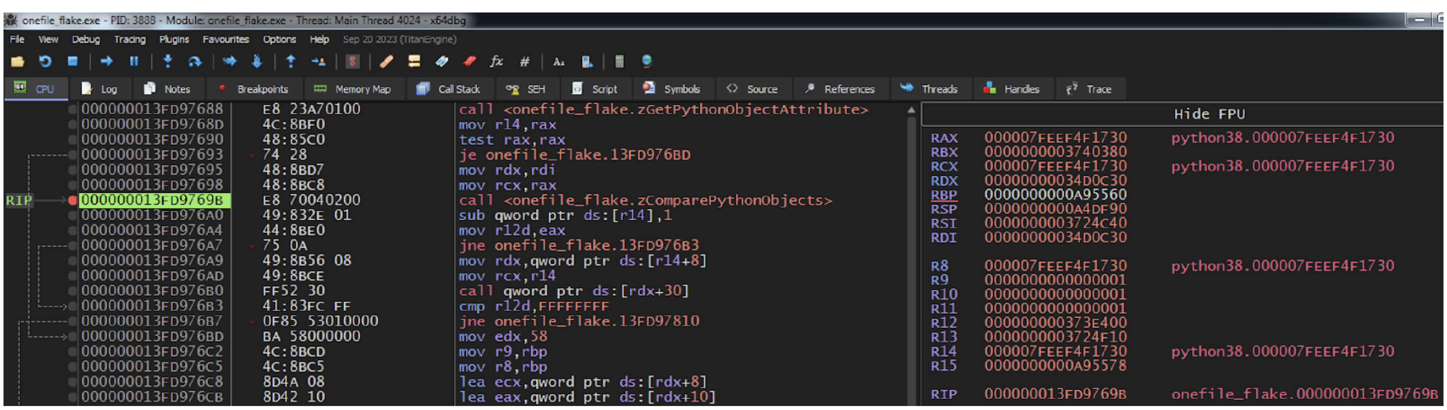

Figure 31: Viewing program state at breakpoint in x64dbg

Optionally, we can confirm that the second argument to the function `zComparePythonObjects` is the Python object that stores our final score. We trick the program into converting the Python object into a C long by forcing it to call the CPython function [PyLong_AsLong](#) and passing our target object as the first argument. We resolve the address of PyLong_AsLong using the x64dbg symbols window:

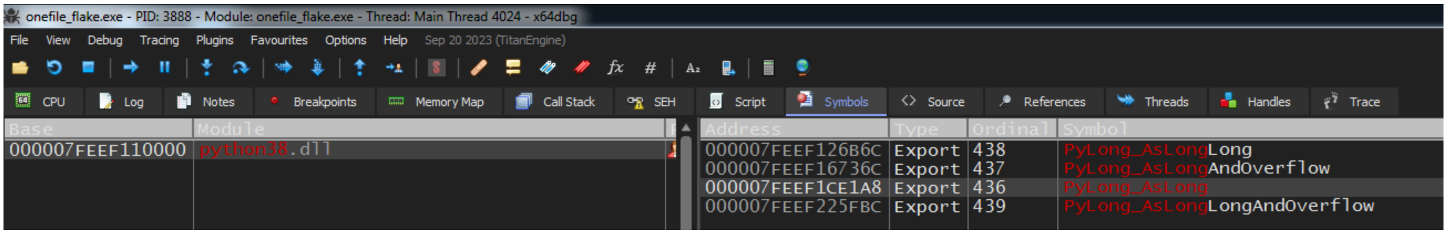

Figure 32: Resolving PyLong_AsLong address using x64dbg

We set the register RIP to the address of PyLong_AsLong, move the value stored in the register RDX to the register RCX, and select Debug > Execute till return. We see that register RAX contains the value 0x4E20, or 20,000, our final score just as we expected.
We restart the program, repeat our previous steps to hit the breakpoint at the address 0x14048769B (0x13FD9769B after rebasing), and step over the function call. We see that the register RAX contains the value zero:

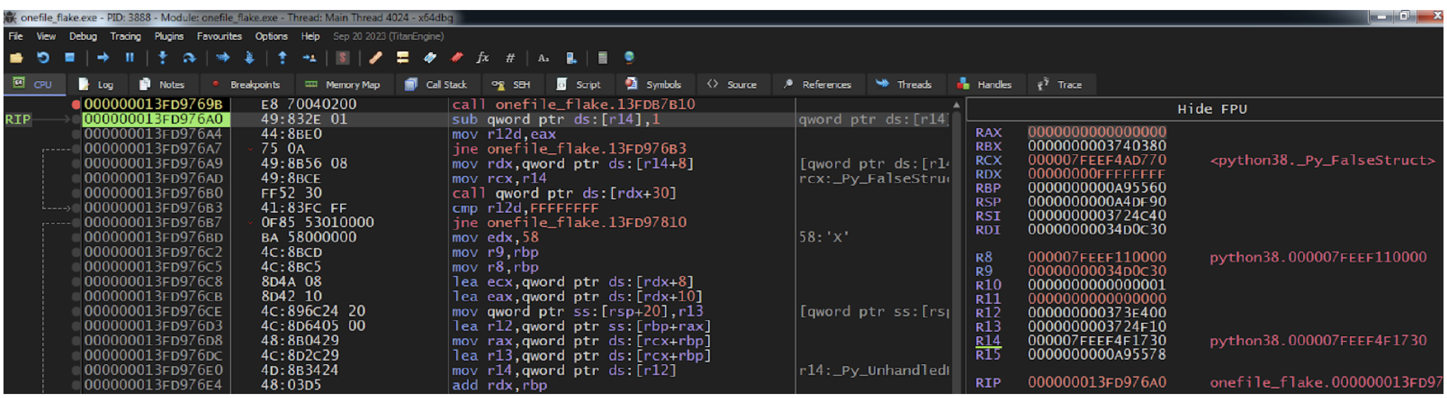

Figure 33: Viewing return value in x64dbg

We change the value stored in the register RAX to one which forces the function `FUN_13FD975F0` to return a True Python object and see that the game accepts our score!

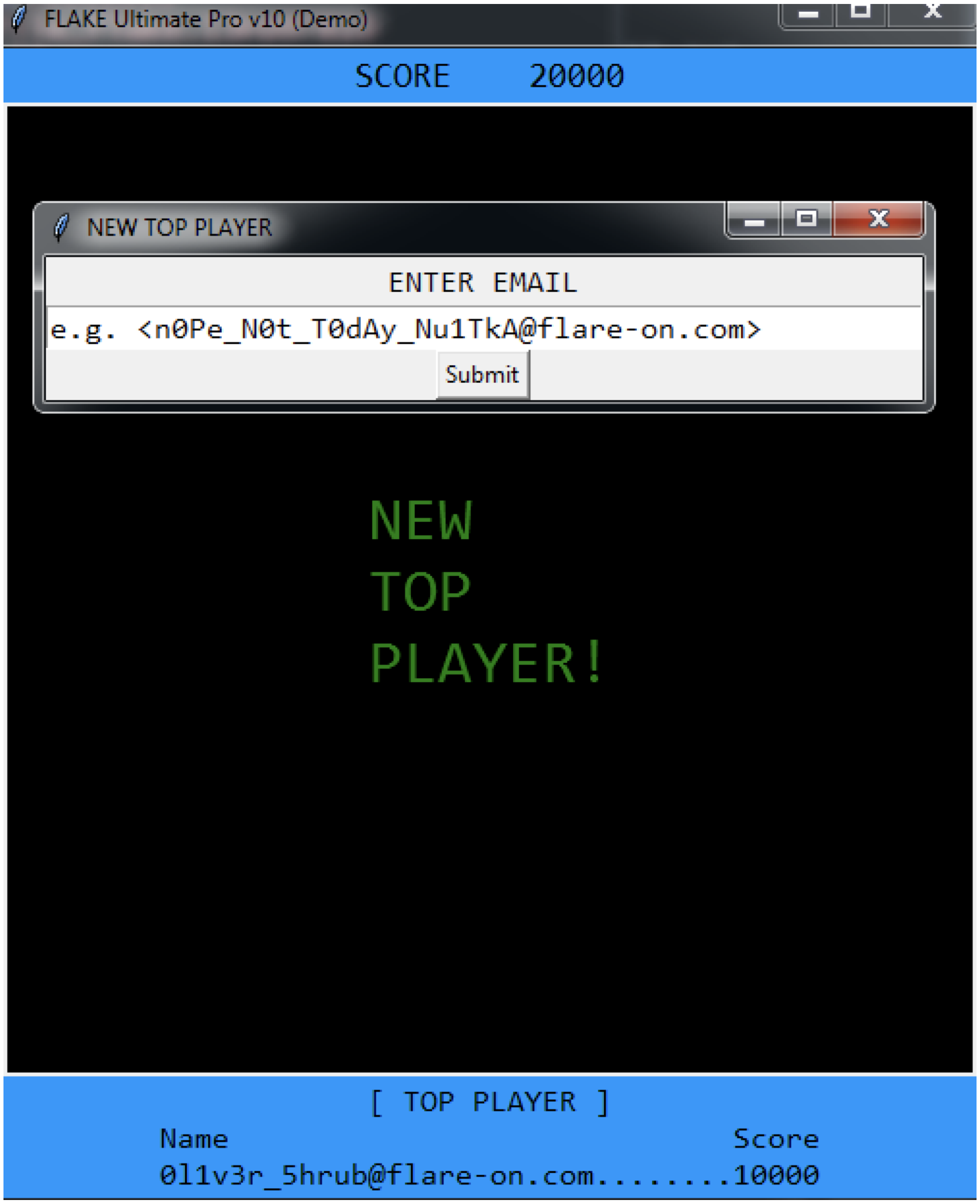

Figure 34: Viewing Flake "NEW TOP PLAYER"

We have beat Oliver Shrub's high score and retrieved the flag:

n0Pe_N0t_T0dAy_Nu1TkA@flare-on.com