Analyzing Emotet with Ghidra — Part 1

medium.com/@0xd0cf11e/analyzing-emotet-with-ghidra-part-1-4da71a5c8d69

Cafe Babe April 22, 2019

Cafe Babe

Apr 19, 2019

·

6 min read

This post I'll show how I used Ghidra in analyzing a recent sample of Emotet.

If you have read this, here is Part 2.

SHA256:

The analysis is done on the unpacked binary file. In this post I'm skipping how I unpacked the file, since what I primarily want to show is how I used Ghidra's python scripting manager to decrypt strings and API calls.

Some short descriptions:

What is Ghidra?

> It is an open source reverse engineering tool suite. You can find out more here —

Why Emotet?

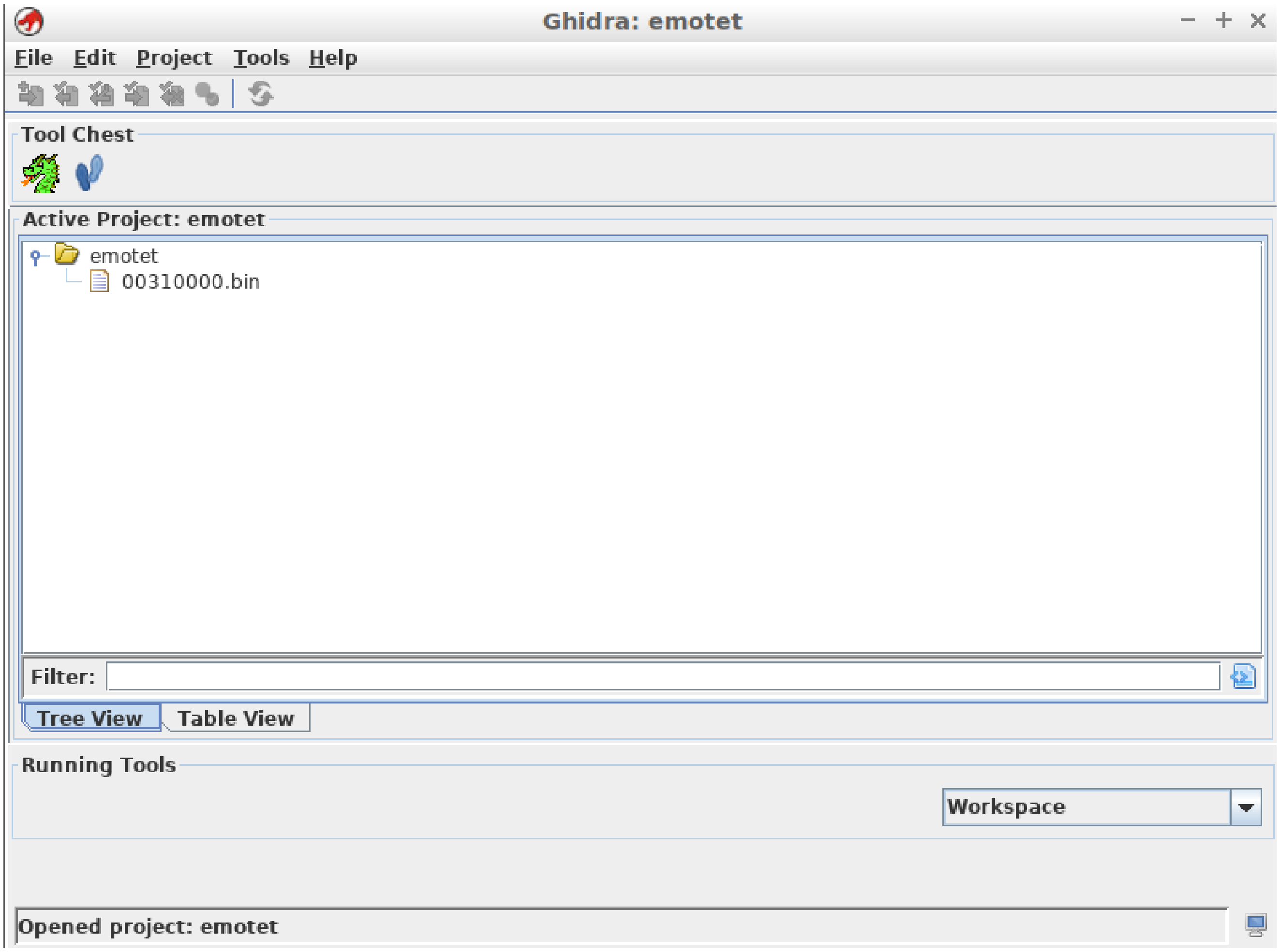
> Emotet is a prevalent malware. Started out as a banking trojan. It is persistent and keeps evolving its infection mechanisms. There are other existing analyses done. A search can lead you there —

Why Ghidra and Emotet?

- For starters, I am looking for a new gig (a.k.a unemployed) and hence cannot afford an . Plus I want to continue being a Malware Analyst.
- Using the free version is still amazing, but I miss not being able to use IDA Python. I did use IDA's own scripting language IDC but…I like python. Implemented just one of the functions of Emotet .

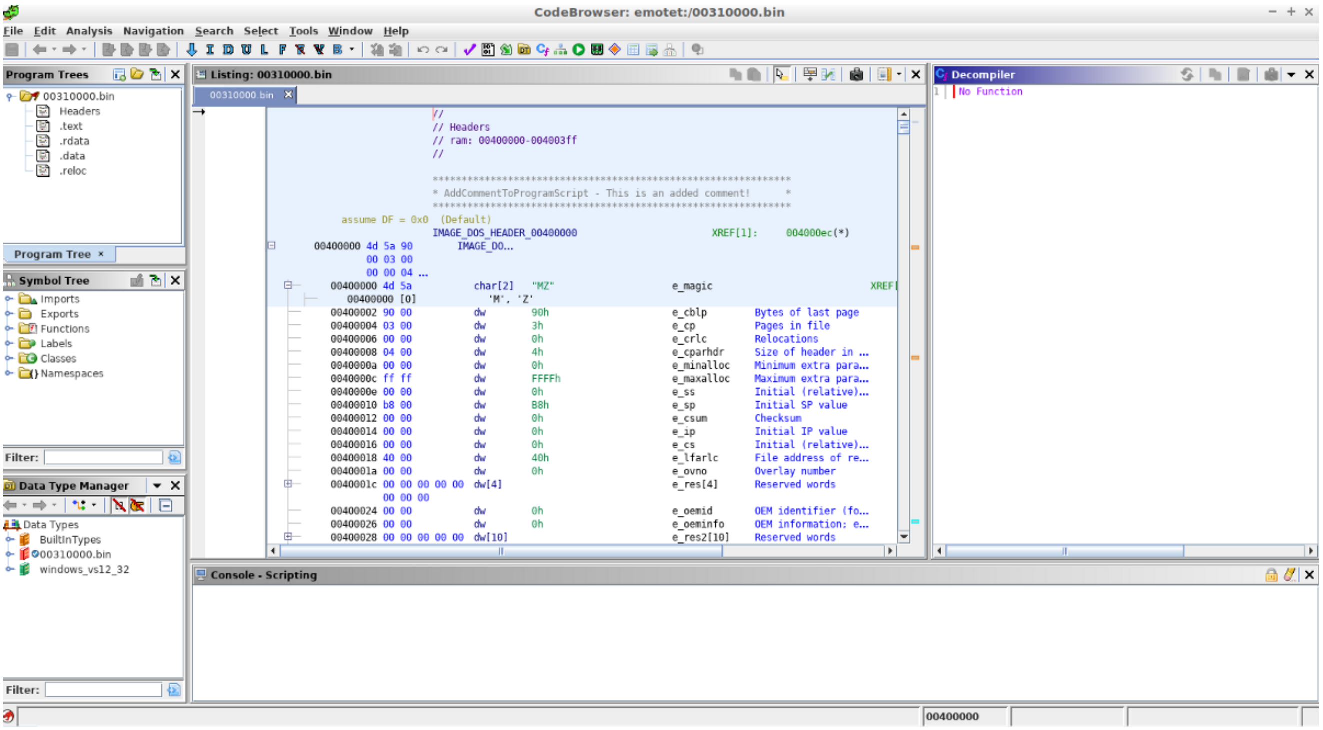
Opening up Emotet with Ghidra

Ghidra is about creating projects. Following the on-screen instructions, I created a project named “Emotet”. To add files to analyze into the project, simple type or go to .



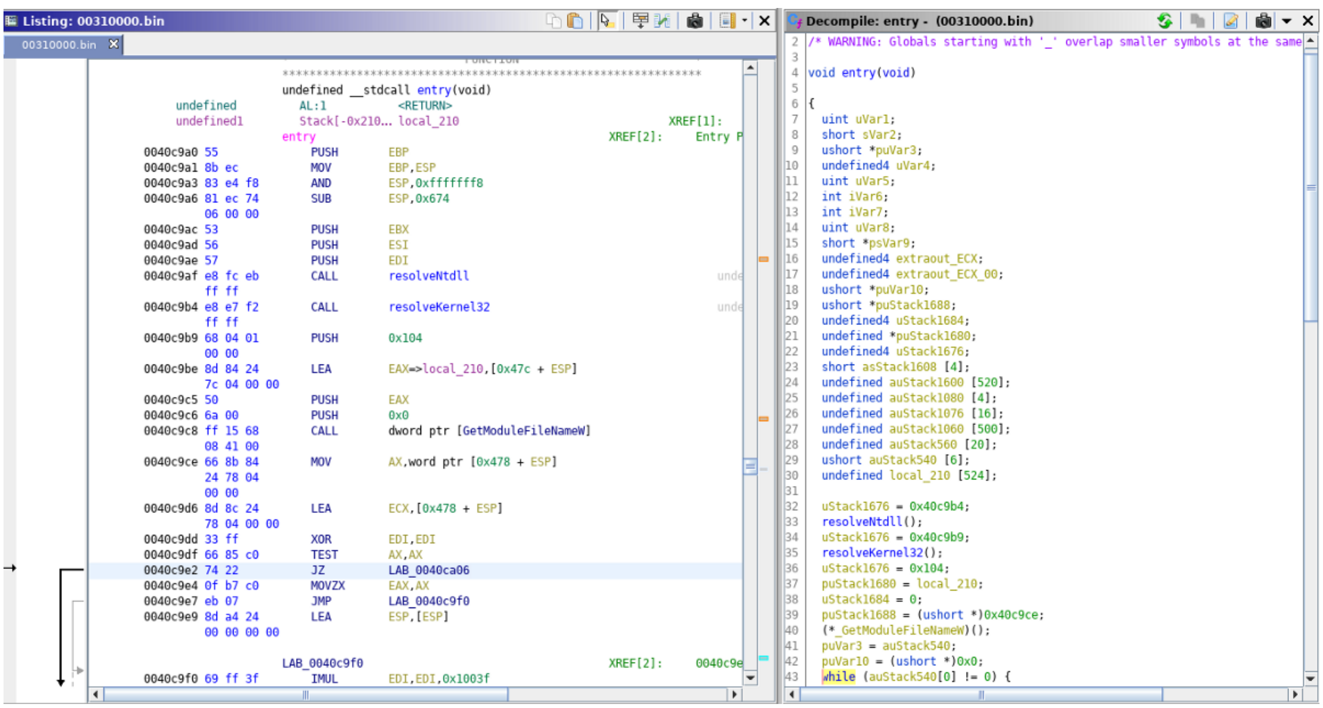
1. Imported Emotet binary

Ghidra displays properties regarding the file that gets imported. Double click on the file name and it opens it up in CodeBrowser which is a tool that disassembles the file.



2. Emotet view in CodeBrowser

Under the Symbol Tree (usually on the left or you can go to), I filtered for “entry” to get to the binary’s entry point.



3. Entry Point of Emotet

Under Listing we see the compiled code and on the right is its decompiled code. Since I’ve already analyzed these binaries, some of the sub routine calls and offsets in these images will have been renamed by me. To rename an offset, right-click an offset value and select (or type).

Emotet’s Function Calls

Emotet encrypts its strings and stores its API call names as hashes. So statically viewing this file, is a pain to read.

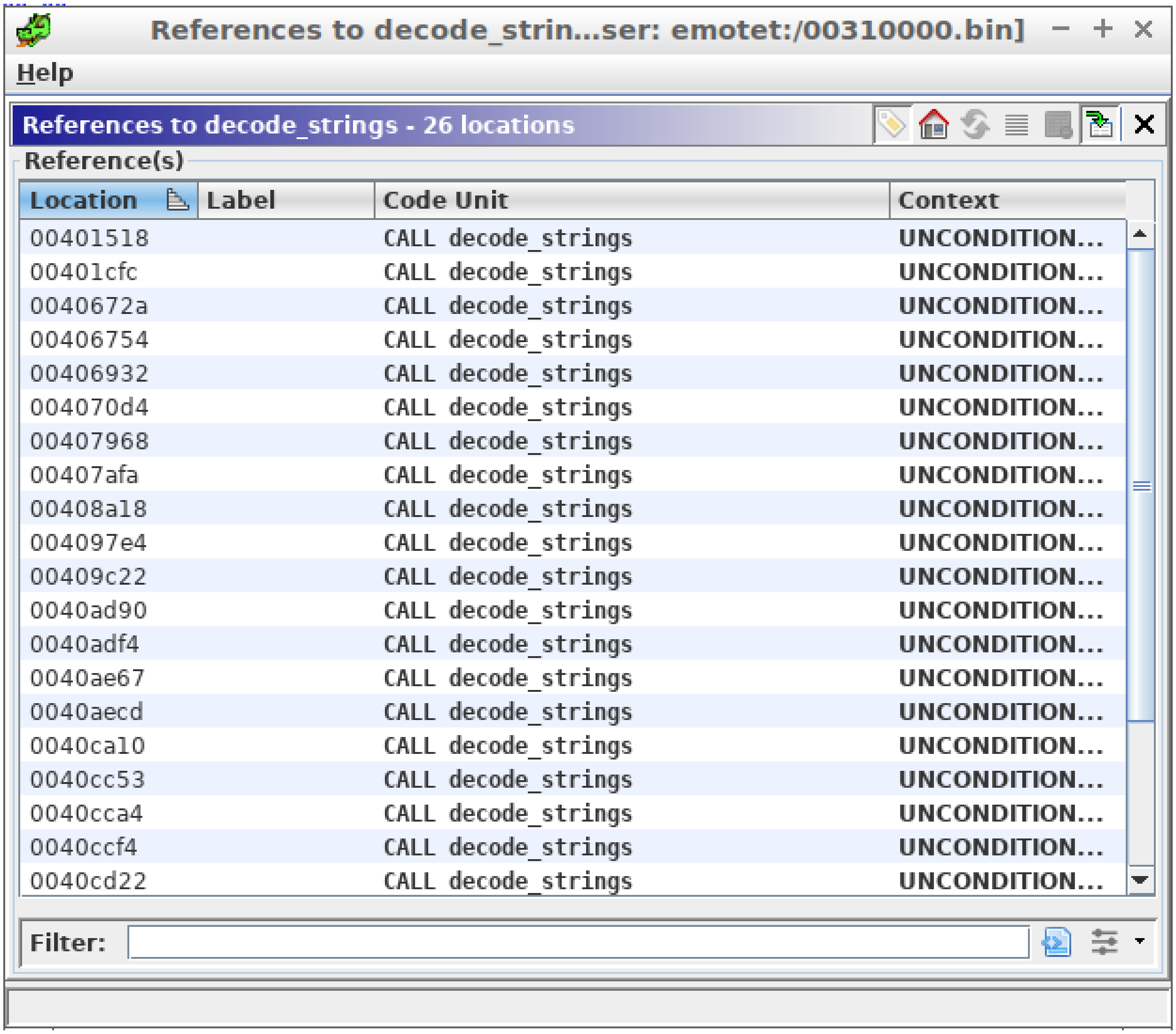
Without going into much detail about Emotet’s payload (that would require another blog entry), I will show how to make this binary a bit more easy to follow. It does require to initially go through each function and figure out the math (possibly using , or whichever debugger so to make it a little less painful).

In this case I wanted to figure 2 methods used by Emotet. The first function is a simple xor routine that it uses to decrypt strings. It looked deceiving complex (because of the use of shift operators in the function), only till after running one iteration in that I realized what was happening… . The second function finds which API name matches which hash (I will cover this in Part 2). This I felt was a bit more clever, but still easy to understand after running in .

Then using Ghidra’s Script Manager, I’ll show how I implemented the python scripts to decrypt the strings and resolve the API calls used in the binary.

How are the Strings encrypted?

In the binary, I’ve noticed a lot of references to the function call at . This call decrypts for the strings. I renamed it to . To find references made to the function, right click the function and select .



4. References to decode_strings

```
                       LAB_0040ca06                              XREF[1]:     0040c9e2(j)
    0040ca06 ba 3e 5a       MOV        EDX,0x77265a3e
             26 77
    0040ca0b b9 e0 fc       MOV        ECX,DAT_0040fce0                                 --%x
             40 00
    0040ca10 e8 5b 51       CALL       decode_strings                                   ecx = offset
             ff ff                                                                      edx = key
    0040ca15 8b f0          MOV        ESI,EAX
```

5. Call being made to decode_strings

The function takes in 2 arguments that are stored in and (Image 5). is the offset of the encrypted string. is the xor key. The decrypted string gets stored in memory allocated in the heap and the address gets passed to .

(Side Track: I have added the string "*ecx = offset \n edx = key*" as a repeatable comment to the function. Right click the address and select or type)

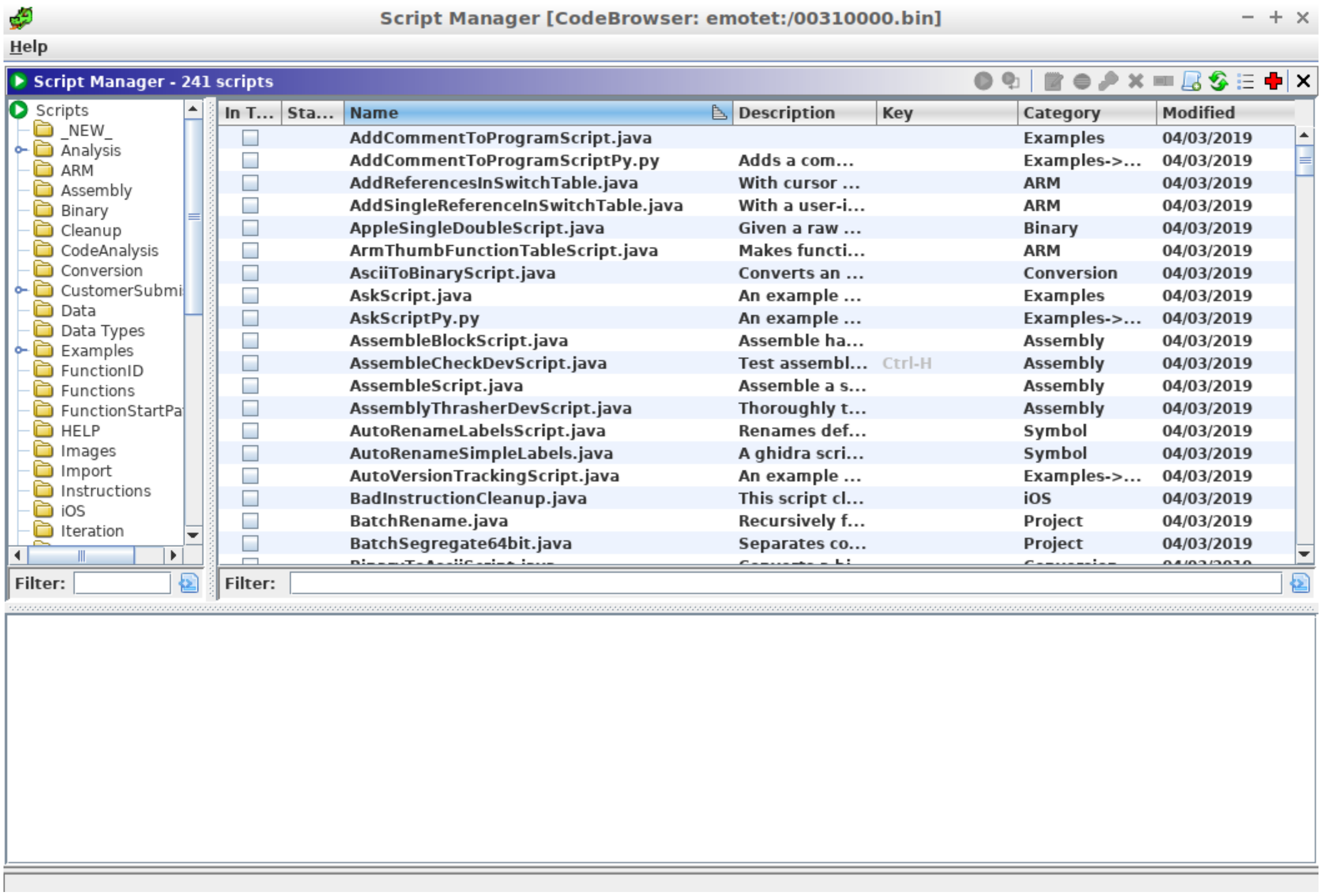
The first dword at the offset xor'ed with the key returned the length of the string. The next subsequent set of dwords were xor'ed up until the string's length.

Now for the more exiting part, automating this with a python script in Ghidra.

Using Python to Automate Decryption

6. Script Manager Icon

In the top toolbar section of Ghidra, we see this icon in image 6. It takes us to the Script Manager. Else you can select .

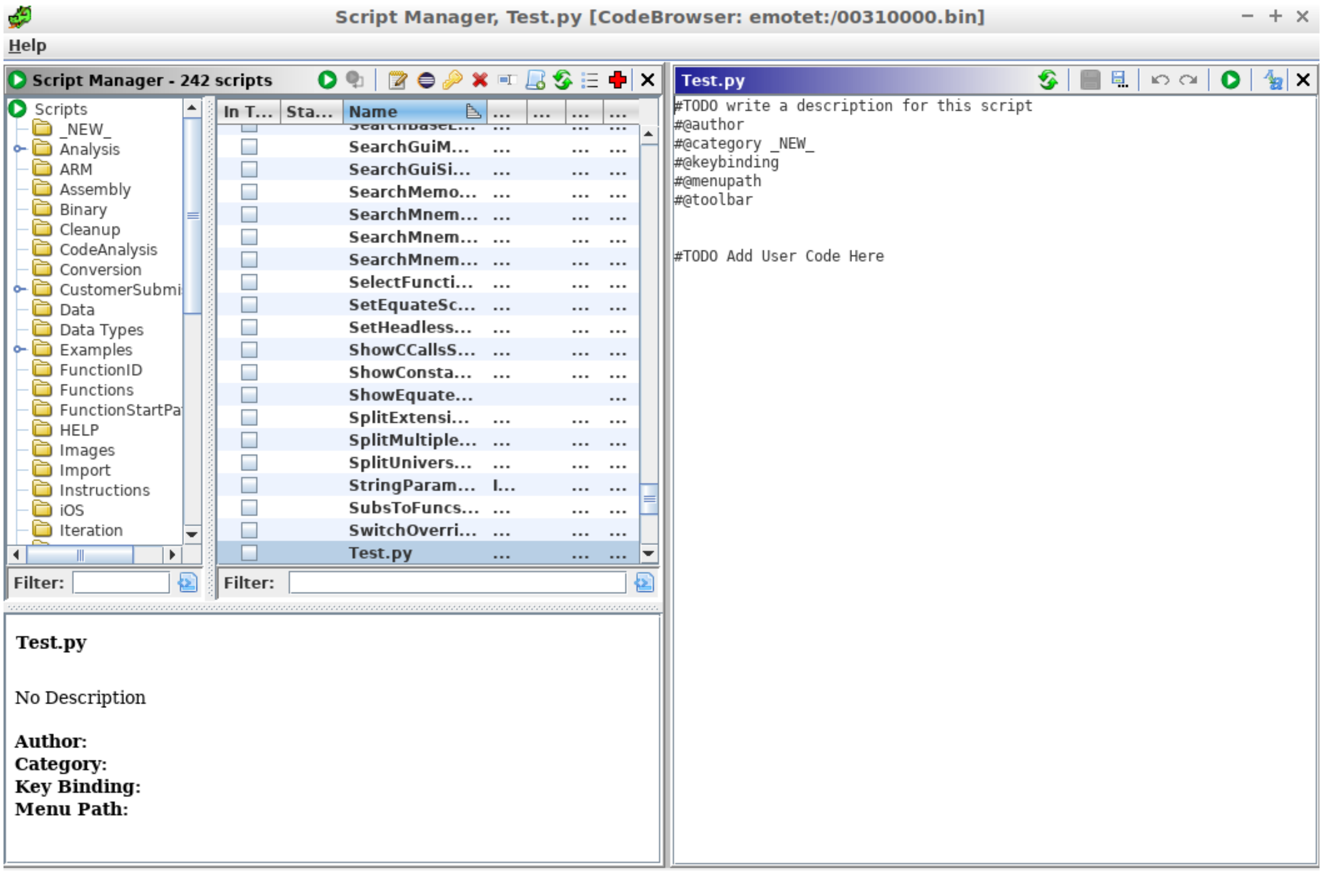


7. Script manager

The Script Manager displays a list of scripts written in either Java or Python. They come with the installation. The script manager also has some python script examples. So, I filtered for .py scripts to help me understand how to proceed in writing a python script. The Python Interpreter interacts with Ghidra's Java API through Jython. The documentation on the Java APIs provided can be found in a zipped file in the docs directory of your Ghidra installation.

8. Create new script icon

To create a new python script, select this icon — image 8. Select Python and enter a name you'd like to give to your script.



8. A sample test.py script created

Additionally, going through the help docs (under)and reading under , there is a description of the metadata tags that gets generated when creating a new script.

I've uploaded the script into my github repo and you can follow it here — https://github.com/0xd0cf11e/ghidra/blob/master/ghidra_emotet_decode_strings.py

```
MOV      ECX,DAT_00410130                     %s\%s

CALL     decode_strings                       ecx = offset
                                              edx = key
```

9. Decrypted string displayed as comment

The idea behind the script is to display the strings that get decrypted as comments next to the instruction where its offset is moved to (Image 9).

```
                      DAT_00410130
00410130 24              ??          24h     $
00410131 11              ??          11h
00410132 23              ??          23h     #
00410133 1d              ??          1Dh
00410134 25 73 5c        ds          "%s\\%s"
         25 73
```

10. Bytes patched in the binary.

And as well to patch the bytes in the binary (Image 10).

```
# get all code references made to the function
refs = getReferencesTo(toAddr(loc))
```

First step, I wanted to find all the code references made to the function.

```
# The parameters passed to the decode function
# are in registers ecx and edx
# iterate through max 100 instructions
# to search for the values moved to the register
i = 0 # counter
ecx = 0 # offset with data
edx = 0 # xor key
comm = 0 # offset to comment on
while((i < 100) and ((ecx == 0) or (edx == 0))):
        inst = getInstructionBefore(inst)
        if "MOV ECX" in inst.toString():
                comm = inst.getAddress()
                ecx = inst.getAddress(1)
                print("ECX = %s" % ecx)
        if "MOV EDX" in inst.toString():
                edx = getInt(inst.getAddress().add(1))
                print("EDX = %s" % edx)
        i += 1
```

Iterating through each reference, the next step was locating for the opcode instructions and . The instructions weren't always immediately before the call to the function. So I iterated through a max of 100 instructions to search for the opcodes.

After that I was all set to carry out the xor routine and patch the bytes and comment at the instruction offset where was carried out.