Week 16 – Cr3ckMe Solution (hard)

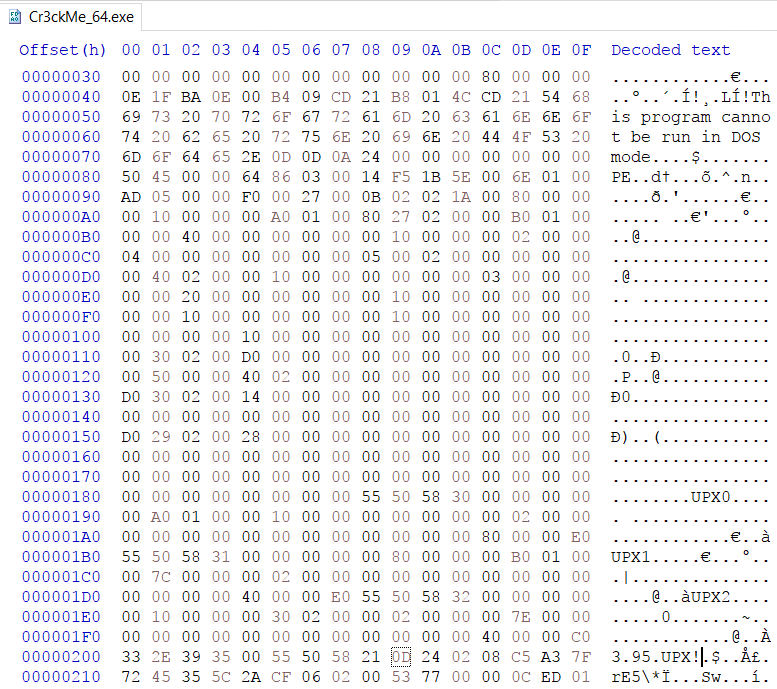
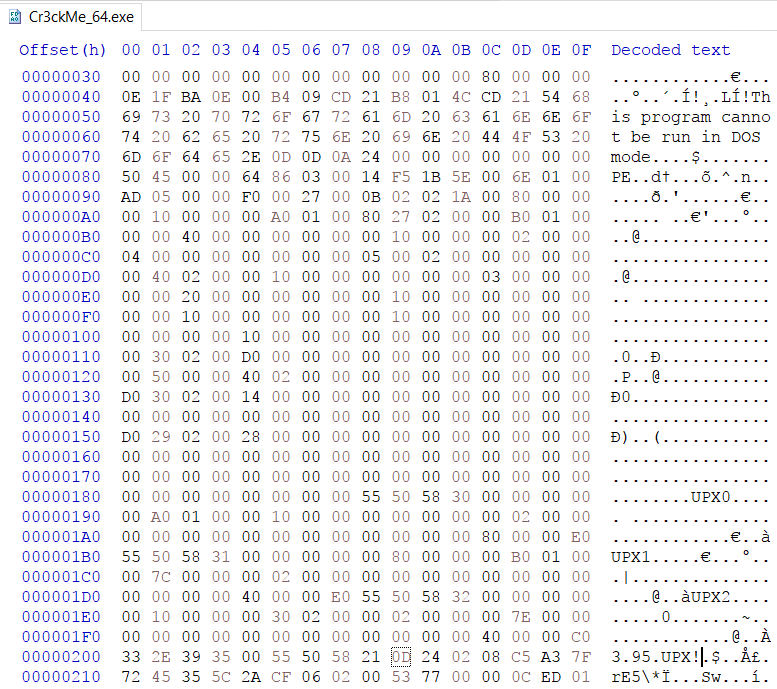
Cr3ckMe (Level Three)

Using x86dbg

Why x86dbg?

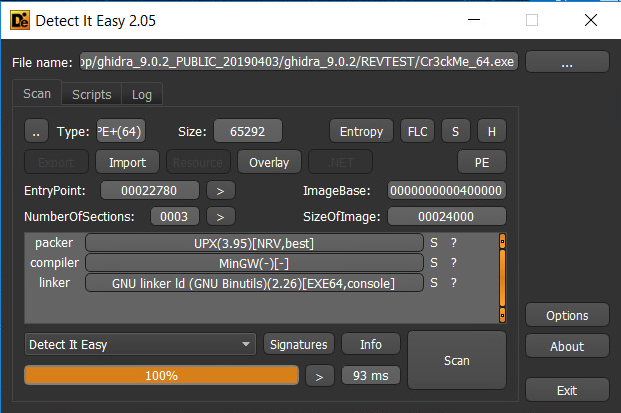
Cr3ckMe.exe is packed by UPX. If we were to put it in Ghidra, we would not be able to find the proper main function. This can be verified by simply looking at the file with a hex editor, or putting it through a PE identifier such as Detect-It-Easy.

**Hex Editor**

****

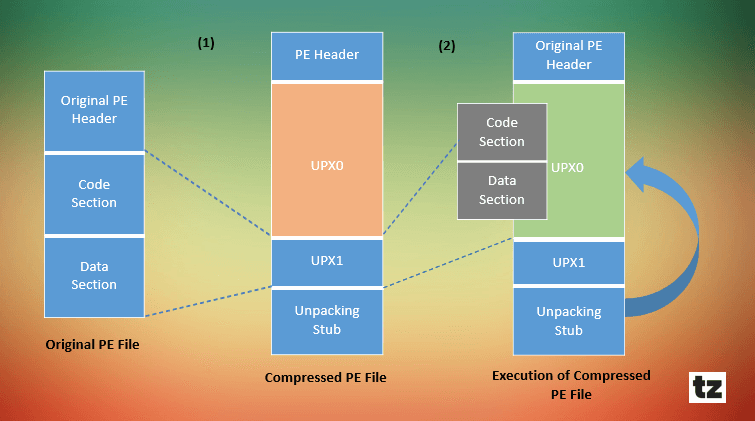
As seen in the screenshots, the UPX header names (UPX0, UPX1, UPX2) are easily distinguished. The UPX version is also clearly seen (3.95).

**Detect-It-Easy**



Detect-It-Easy identifies packer, compiler and linker signatures in a file and displays them. It also has other functions such as finding the entropy of the file, the Entry Point offset and an in-built hex editor (H).

How UPX works and how to unpack it manually

**Packing Process**

UPX compresses all existing sections such as .text, .data, etc. and these sections are named UPX0, UPX1 and so on. It then adds a new code section at the end of the file which will decompress all the packed sections when the file is executed.

When executed the packed .exe [file-:](file:///-)

1. Starts from the end of the UPX packed file (unpacking stub)
2. Saves the current Register Status using PUSH instructions
3. All the packed sections are unpacked in memory
4. Resolves the import table of the original .exe file
5. Restore to original Register Status using POP instructions
6. Jumps to Original Entry Point (OEP) to begin actual execution

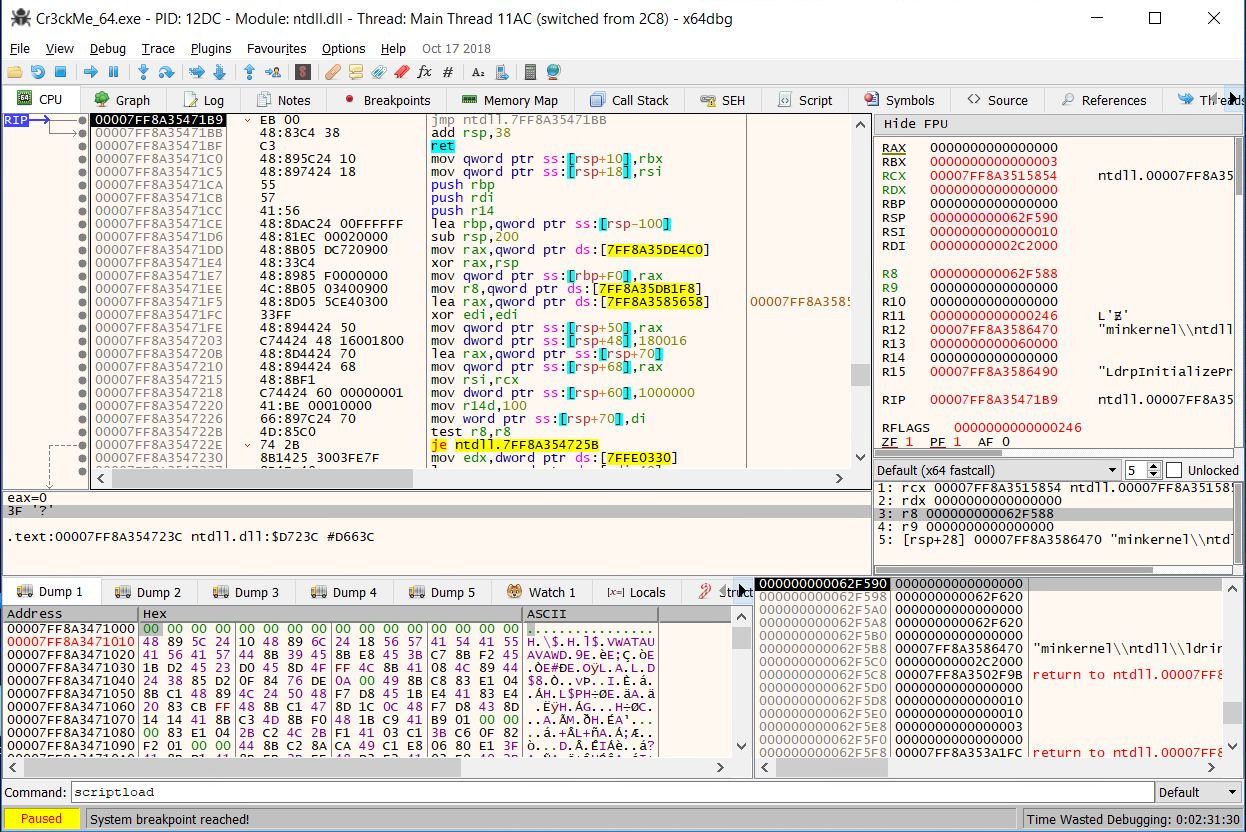
**Manually Unpacking**

By understanding the packing process, we understand how to unpack and extract the executable. We simply need run through the code until it decompresses the packed sections and jumps to the OEP and extract the unpacked executable from there.

Since we require to run through the packed executable and extract a section from it, we use x64dbg as our debugger and the Scylla plugin allows us to dump the unpacked executable.

Debugging

**Getting Familiar**

Before going into it, we can get familiar with the executable itself by running through it a couple of times.

After the first return, we stop at the EntryPoint of the Cr3ckMe at 0x422780 with the instruction “PUSH rbx”.

Remember that since the executable is packed, this is a newly generated entry point made by UPX and not the actual entry point of the Cr3ckMe program.

If we run it again, the Cr3ckMe closes.

**Patterns to Identify OEP**

To unpack UPX, there are several patterns that we can observe in the assembly code of the program to help us identify key parts of the program.

1. At the beginning of every UPX program (Entry Point), there is a PUSHAD instruction.  
   [However, since this is a 64-bit instruction, it pushes all the registers i.e. rbx separately]

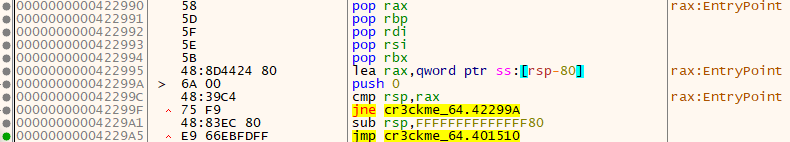
If we look back at the EntryPoint of the packed executable,  
we can see that the first instructions of the program are push instructions. Since PUSHAD is a 32-bit instruction and we are working with a 64-bit executable, these push instructions are the 64-bit equivalent of the PUSHAD instruction. This tells us we are on the right track.

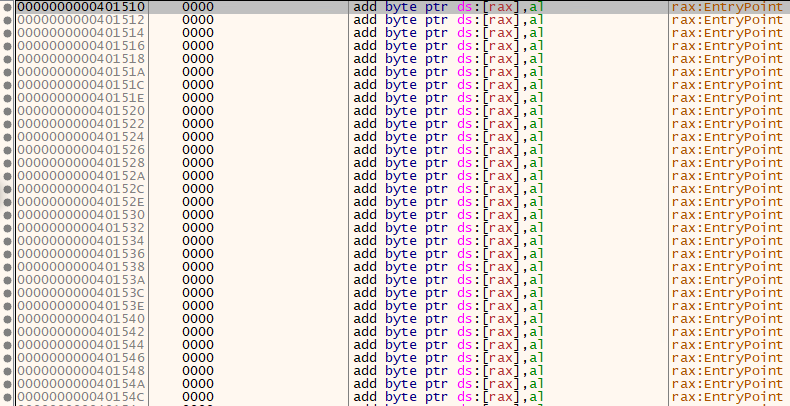
1. There are several standard instructions before the jump to the OEP that was observed when manually unpacking 32-bit packed executables, usually at the bottom of the readable disassembly (to decompress the packed code):
   * “POPAD” instruction
   * “LEA <register>,qword” instruction
   * “PUSH 0” instruction
   * “CMP esp,eax” instruction
   * “JNE <address>” instruction
   * “sub esp,<value>” instruction
   * “JMP <address of original entry point>” instruction

**Finding the Original Entry Point (OEP)**

By scrolling up from the bottom of the main module (with the Entry Point), we can look for the standard set of instructions that will lead us to the OEP.

Surely enough, at 0x422990:

We can concur that the OEP lies at address 0x401510. However, if we follow that address in the disassembler:

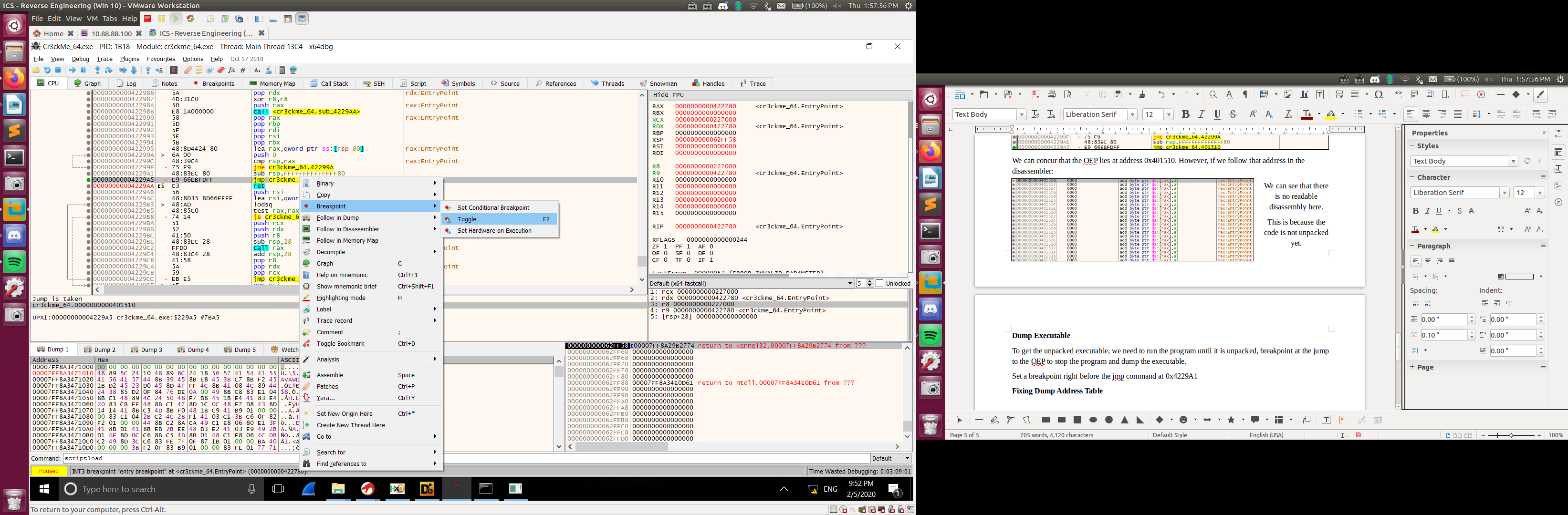
We can see that there is no readable disassembly here.

This is because the code is not unpacked yet.

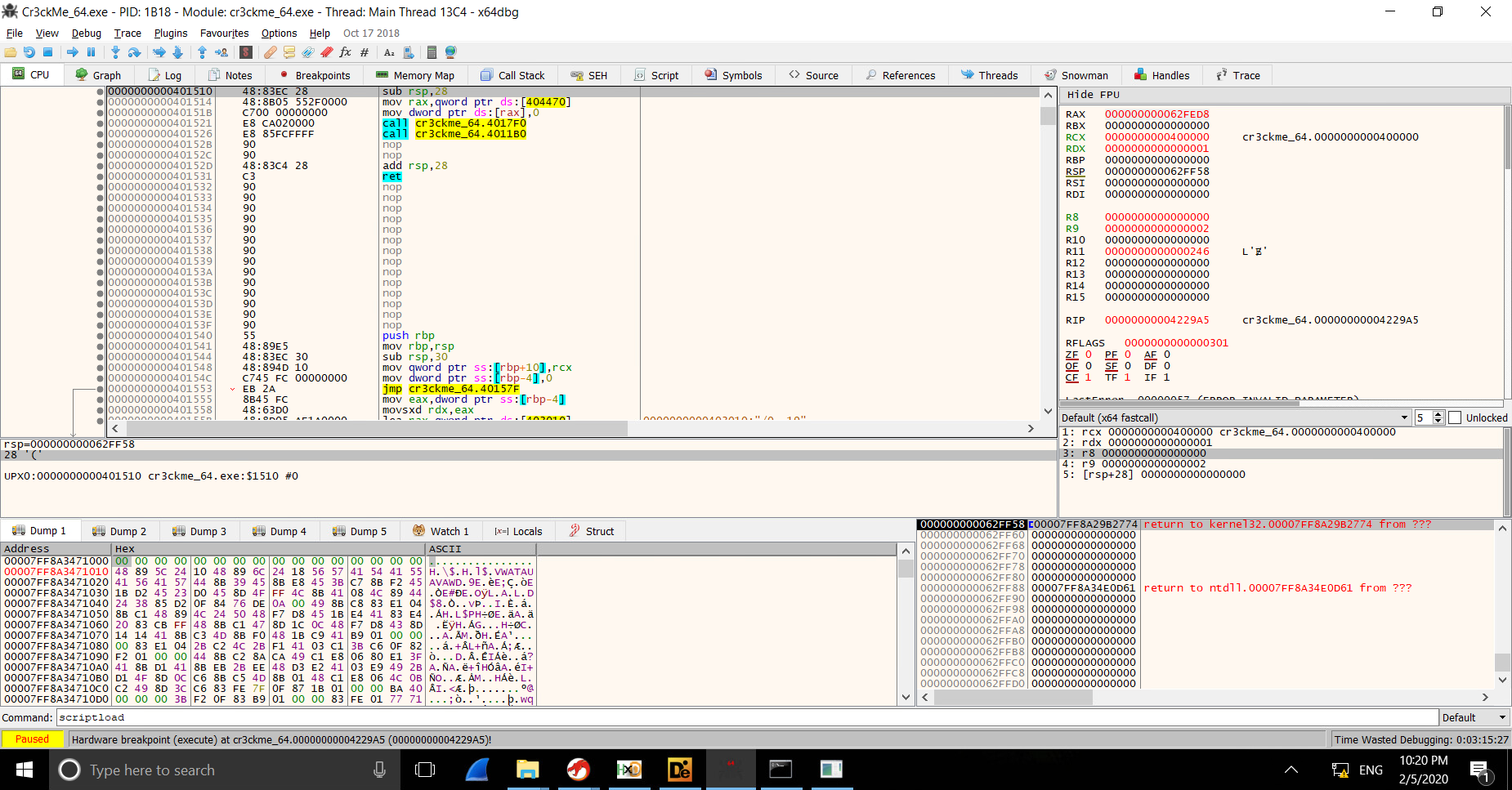
**Manually Unpacking Executable**

To get the unpacked executable, we need to run the program until it is unpacked, breakpoint at the jump to the OEP to stop the program and dump the executable.

Set a breakpoint on the jmp instruction at 0x4229A5:

Once the breakpoint is set, when the program runs over the jmp instruction, it will break and stop before running the instruction. If the breakpoint is set, the address will be highlighted in red and a red circle will be next to the address such as:

Run the program until it breaks at the jmp instruction, then step into the jump.

Now we see the same address as before, but with readable instructions.

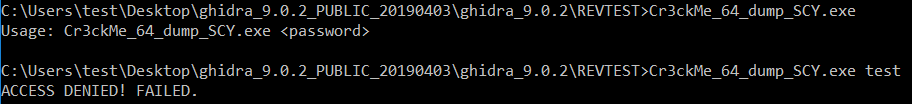
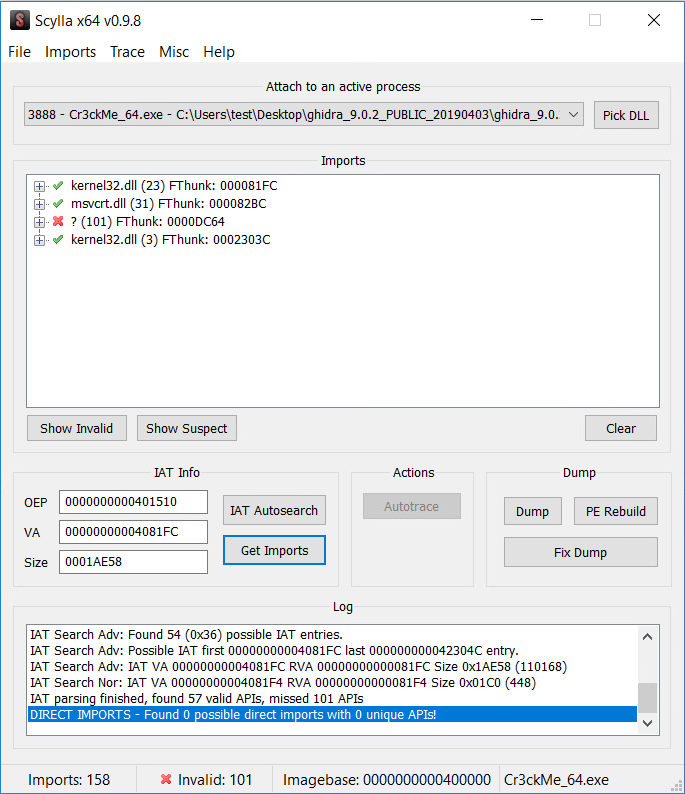
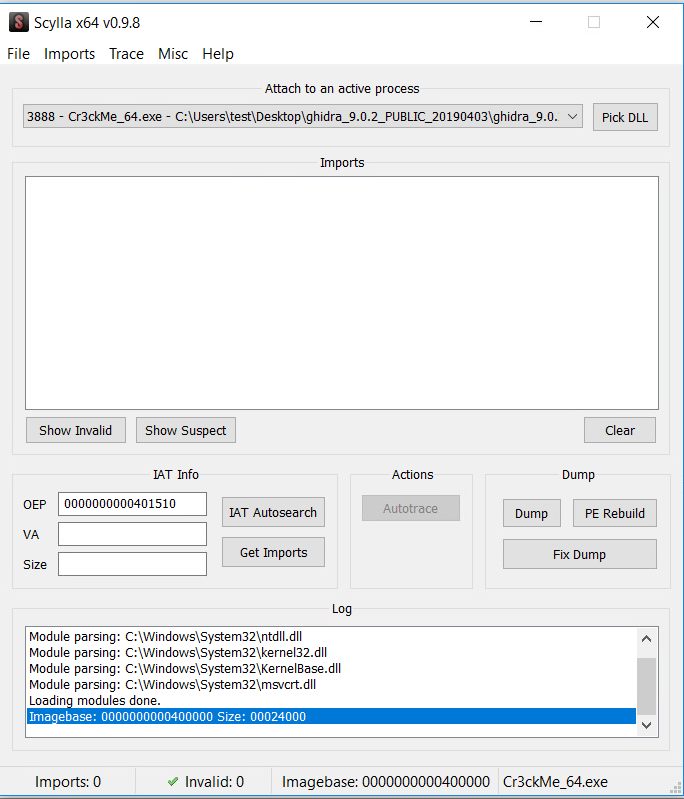
This is the OEP of the executable.  
[0x401510]

From here, we can dump the original code of the executable into an unpacked executable.

**Extracting Unpacked Executable**

x86dbg has a built-in plugin called Scylla, which allows for the attaching and dumping of executables. We will be using this to extract the unpacked executable. Click the Scylla icon in the Toolbar, or go to Plugins > Scylla.

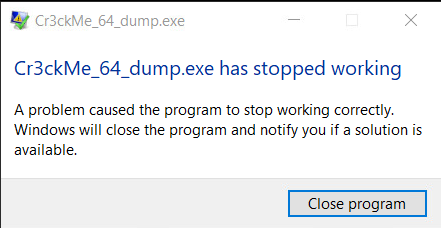
Scylla automatically attaches to the program being debugged by x86dbg and takes the current instruction pointer (RIP) as the OEP. Using Scylla, we can detect the Import Address Table (IAT) of the executable and get the imports list that the executable uses. Scylla simplifies it into simple one-click buttons: (1) IAT Autosearch and (2) Get Imports



>

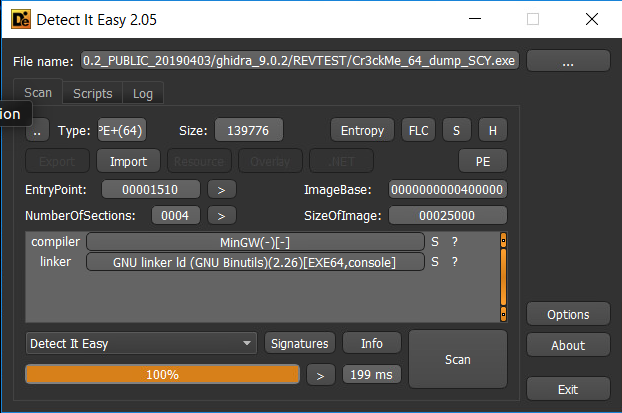
However, there is missing/invalid tree that we need to get rid of before we can dump our executable. We can just right click it and click “Delete the tree node” then click “Dump” to dump the executable.

**Fixing Dump Address Table**

The program will still not work because we need to fix the Import Table which we fetched in Scylla beforehand.

Click “Fix Dump” and select the original dumped executable file. The fixed executable is saved with an appended \_SCY at the end of the filename.

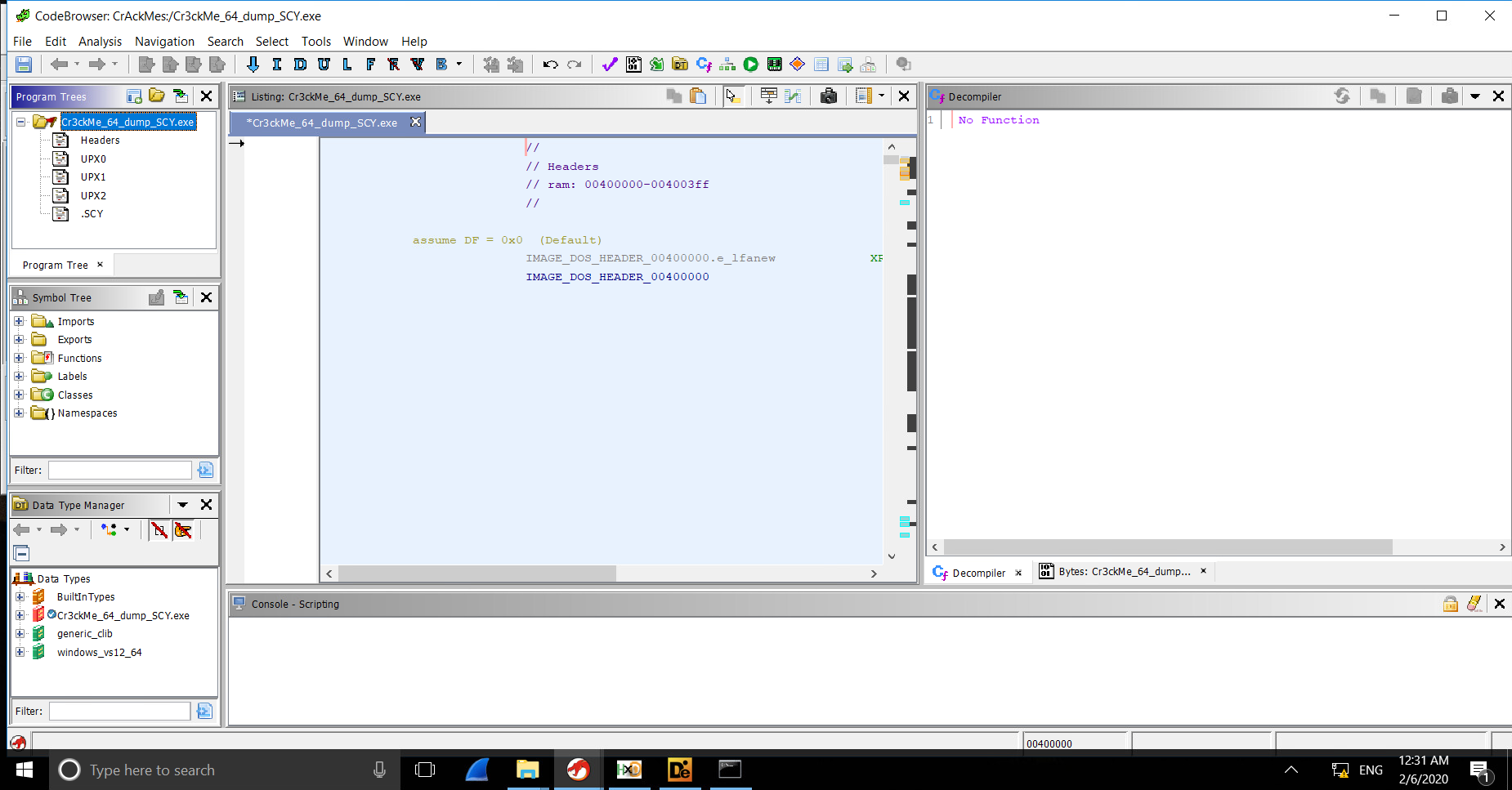
**Unpacked, Now What?**



If we try running the unpacked executable, we can see that it works fine. We can also analyze it with Detect-It-Easy again, and we can see that it is unpacked.

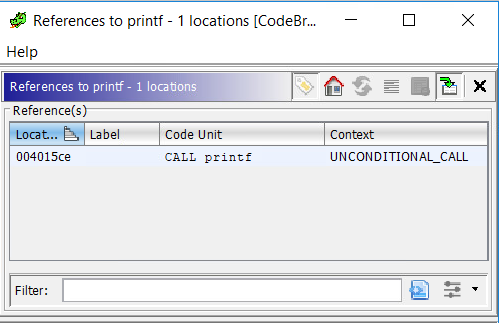
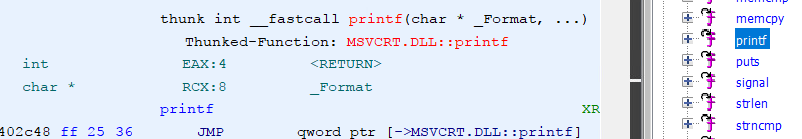
However, we still need to reverse the code. This is where now Ghidra comes in.

Using Ghidra

Finding the main function

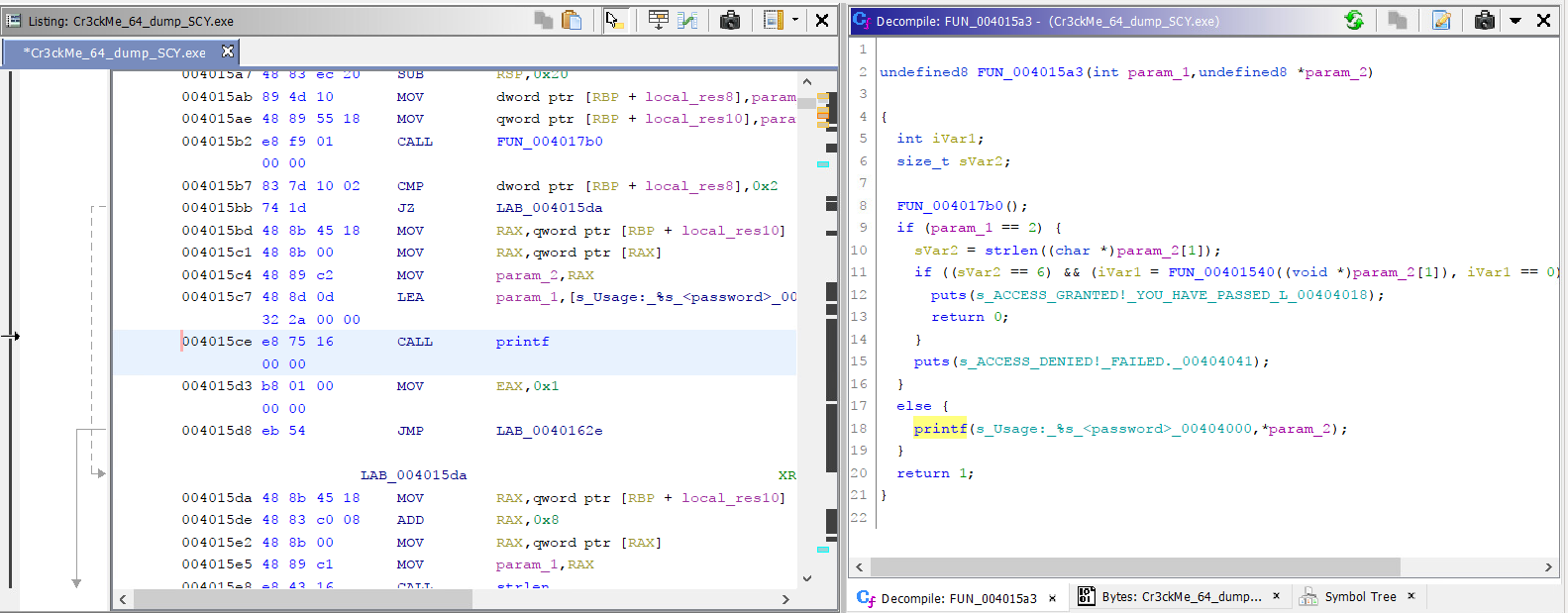
Compared to the original file, we can see more functions (especially C imported functions) under the Symbol Tree, although the main function is still not clear. However, since we know what the program does, we can find the main function by backtracking.

The program prints certain strings depending on the input. Hence, if we can find the printf function and look at functions that call on the printf function (cross-references), it is highly likely that our main function can be found.



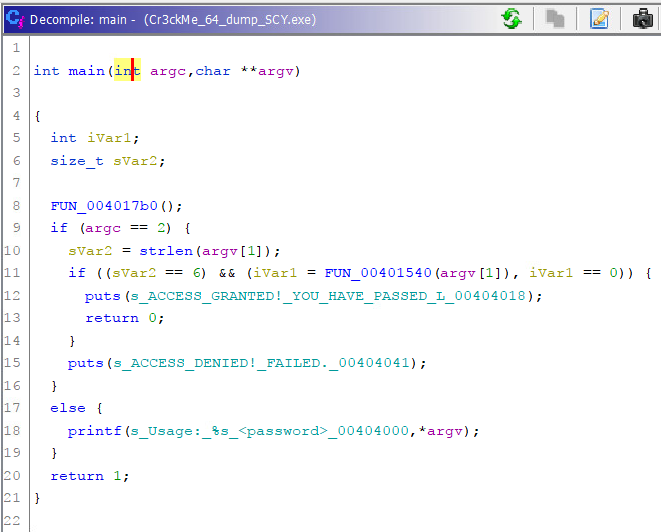
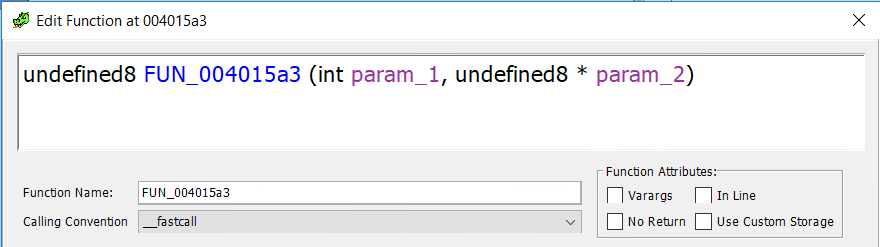
By right-clicking the printf function and clicking “Show references to”, we can look at addresses that call for the printf function. Looking at the References window below, there is only one reference to printf! If printf didn’t work, we could have tried the puts function.

And we’ve found our main function!

Working from the main function

From here, we can input the C standard signature into the function since we know it is a C program from the function names as well as the \_C\_specific\_handler in the MSVCRT.DLL import.

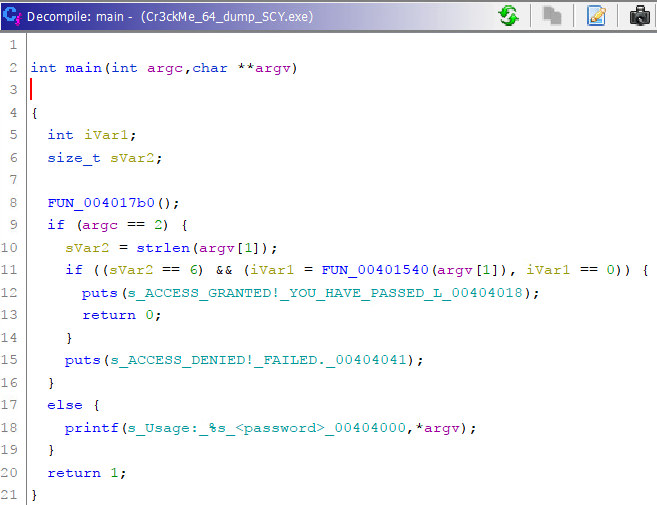
Right-click the first line in the decompile window and click “Edit function signature”:



Then replace the text with “int main(int argc, char\*\* argv)” and the decompile window should look like this.

This makes the program much more readable and easier to understand what is being done.

Identifying End Goal & Conditional Statement

Looking at the decompile window, we can easily identify the end goal and basically, where we want our program to run through.

The string “ACCESS GRANTED! ...” is an indicator of us passing the Cr3ckMe.

Before this line of code, there is an IF statement which is our conditional statement for entering this section of code.

So, if we are able to set this IF statement to true, we can crack the file.

Breaking down the Conditional Statement

**Conditional Statement Breakdown**

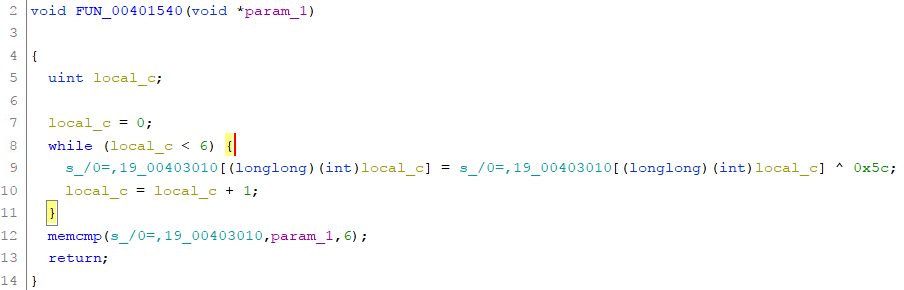
if ((sVar2 == 6) && (iVar1 = FUN\_00401540(argv[1]), iVar1 == 0))

There are two factors that have to be True for this IF statement to be active:

1. sVar2 == 6
2. iVar1 = FUN\_00401540(argv[1]), iVar1 == 0

For the first factor, it tells us that the size of the password is 6 characters, since sVar2 is a size\_t type.

For the second factor, we need FUN\_00401540(argv[1]) to be equal to 0. To do that, we need to know what FUN\_00401540 does to argv[1] to get its value. So, we can double click on the FUN\_00401540 function in the decompiler.

**FUN\_00401540 Function Breakdown**

At first glance, we can notice a few things in this function:

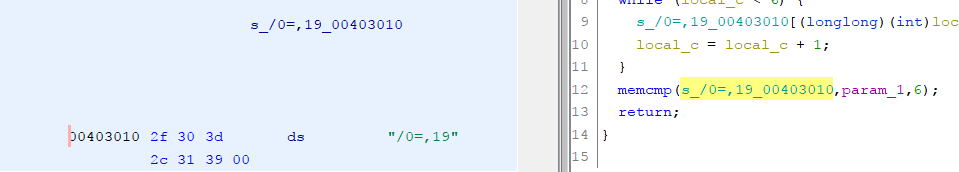
1. a FOR loop (with local\_c variable as a counter) with -:
   1. a string variable (hard-coded variable, in light blue color)
   2. index references to said variable
   3. XOR function against 0x5C
2. memcmp funtion that takes in:
   1. a string variable
   2. additional parameter (password input)
   3. size\_t value of 6

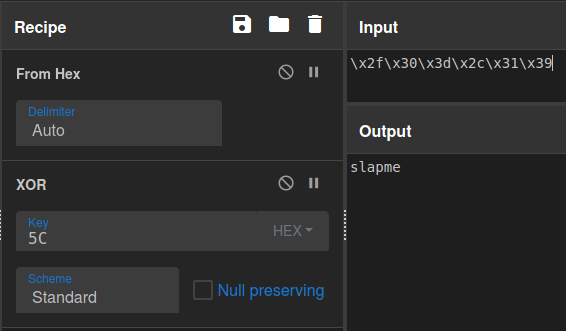
If we were to rewrite the decompiled code into a simpler and easier to understand format, it would look like this:

|  |
| --- |
| for(int counter = 0; counter < 6; counter++){  stringv[counter] = stringv[counter] ^ 0x5c;  }  memcmp(stringv,input\_password,6); |

After simplifying the decompiled code, we can see that this function takes the hard-coded string variable and iterates through each byte and XOR’s it against 0x5C. Then, it gets put into a memcmp function. This string variable should be our password as it is compared against our input.

**Password Found & Deciphered**

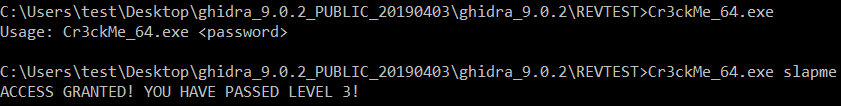
By double clicking on the string variable, it brings us to the supposed “.data” section of the unpacked executable with the value of the string variable which is “/0=,19”. It looks like a garbled mess but this is our password.

Since we know the way the program works, we know the password is stored before being XOR’d and compared.

We can simply XOR the hard-coded value against 0x5c to get the password.

Which is “slapme”.

**Solved**

**References:**

* <https://www.goggleheadedhacker.com/blog/post/6> (unpacking manually)
* <https://securityxploded.com/unpackingupx.php> (explanation of UPX packing and unpacking)
* <https://forum.tuts4you.com/topic/36045-do-you-know-pushad-pushfd-ocmmand-in-x64-mode/> (understanding pushad/popad in x32 and converting to x64)
* <https://www.youtube.com/watch?v=-MZxdGU9BS4> (unpacking upx with x64dbg)
* <https://tech-zealots.com/reverse-engineering/dissecting-manual-unpacking-of-a-upx-packed-file/> (unpacking upx with olly+scylla)