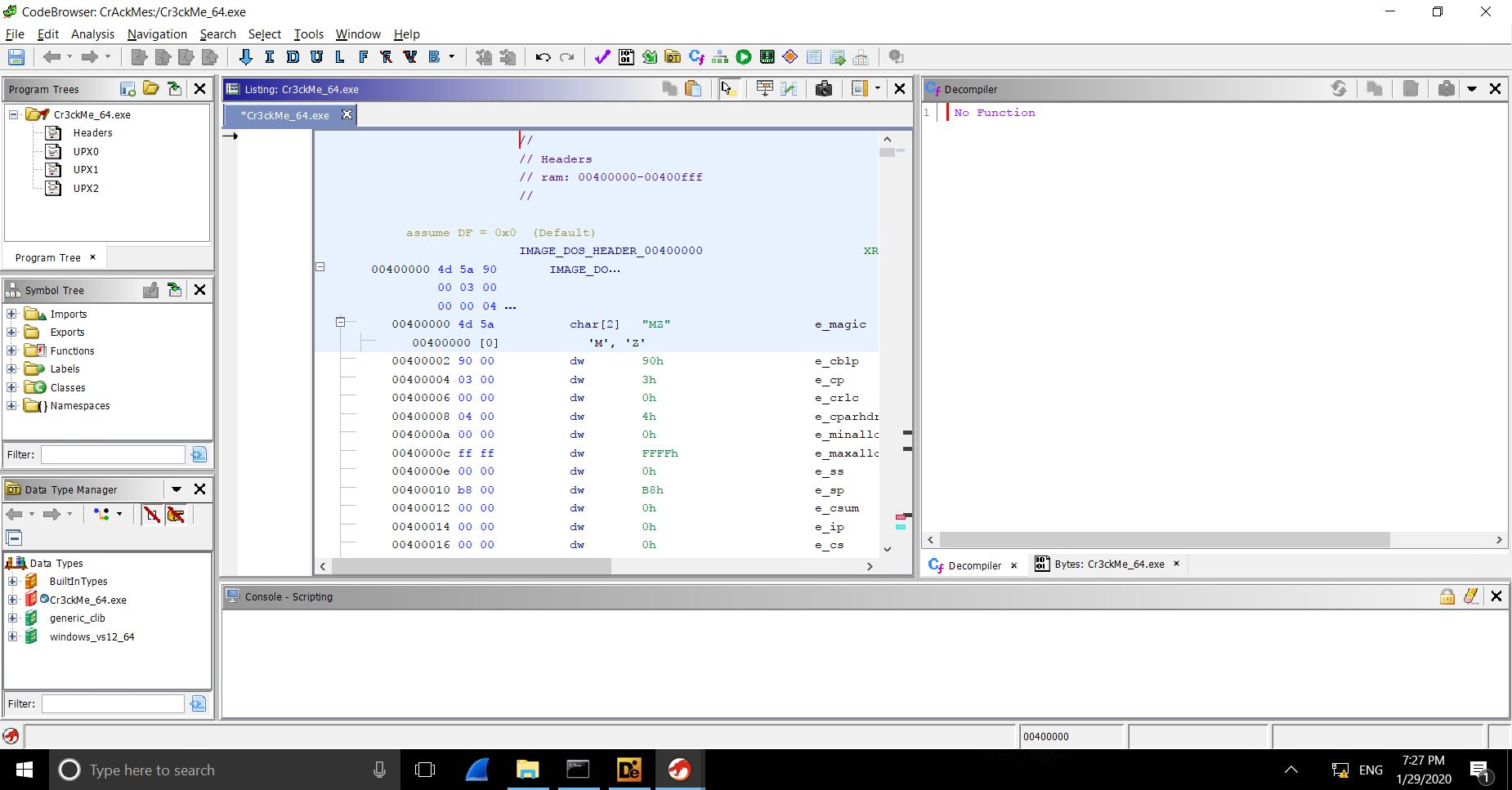
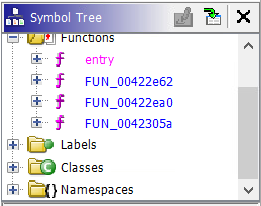
Week 16 – Cr3ckMe Solution

Cr3ckMe (Level Three)

Using Ghidra

What’s this?

Compared to the first two levels of [Cr@ckMe](mailto:Cr@ckMe) files, there is more data written on the header. There is also a different way the program is separated in the headers: Headers, UPX0, UPX1, UPX2.

If we attempt to find the main function now, we can only find the “entry” function and other function names based on their offset.

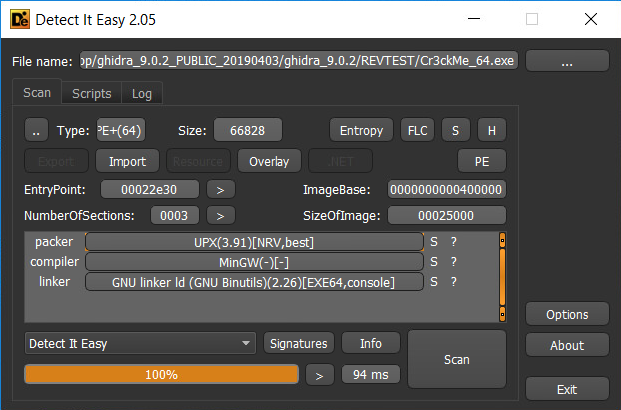
This is a method of **obfuscating** your code, to defend against reverse engineering. The program is packed and by looking at the headers, we can see it is packed by UPX.

Detect-It-Easy

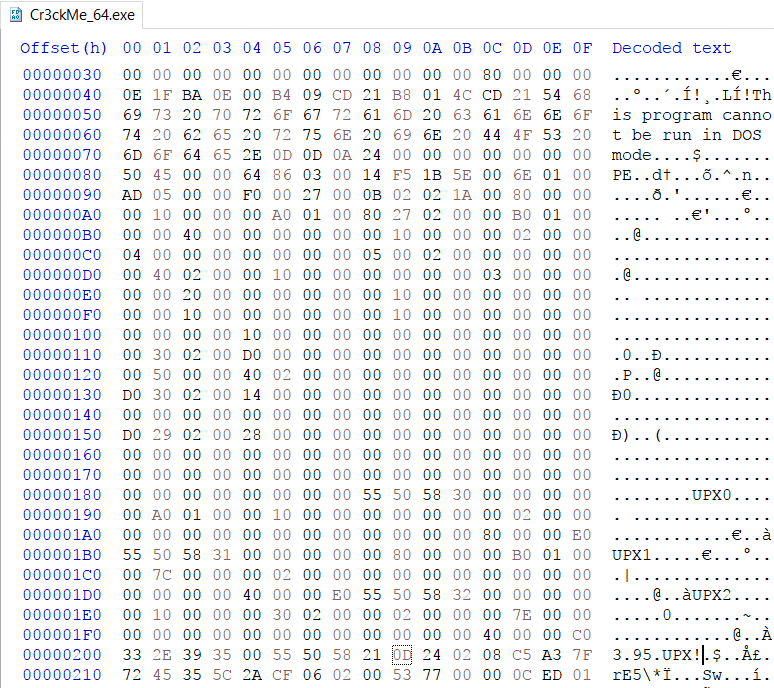
To verify this, we can also use a PE identifier program such as PeiD or DIE, which is what we are going to use. After installing Detect-It-Easy, simply run the program and drag the Cr3ckMe.exe file into the program window.

Using the signatures of common packers, compilers and linkers, DIE will detect the type of each (if present) used in the executable.

It should look like this after dragging the executable over the program window:

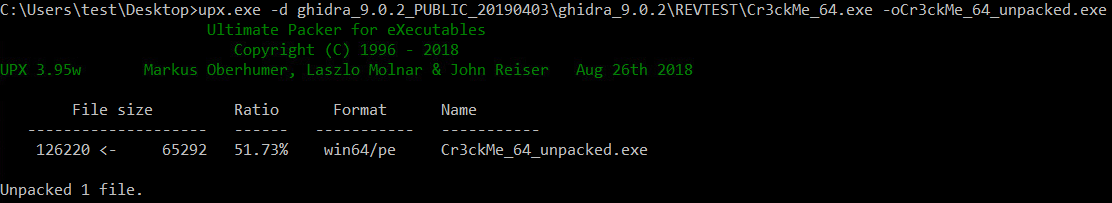
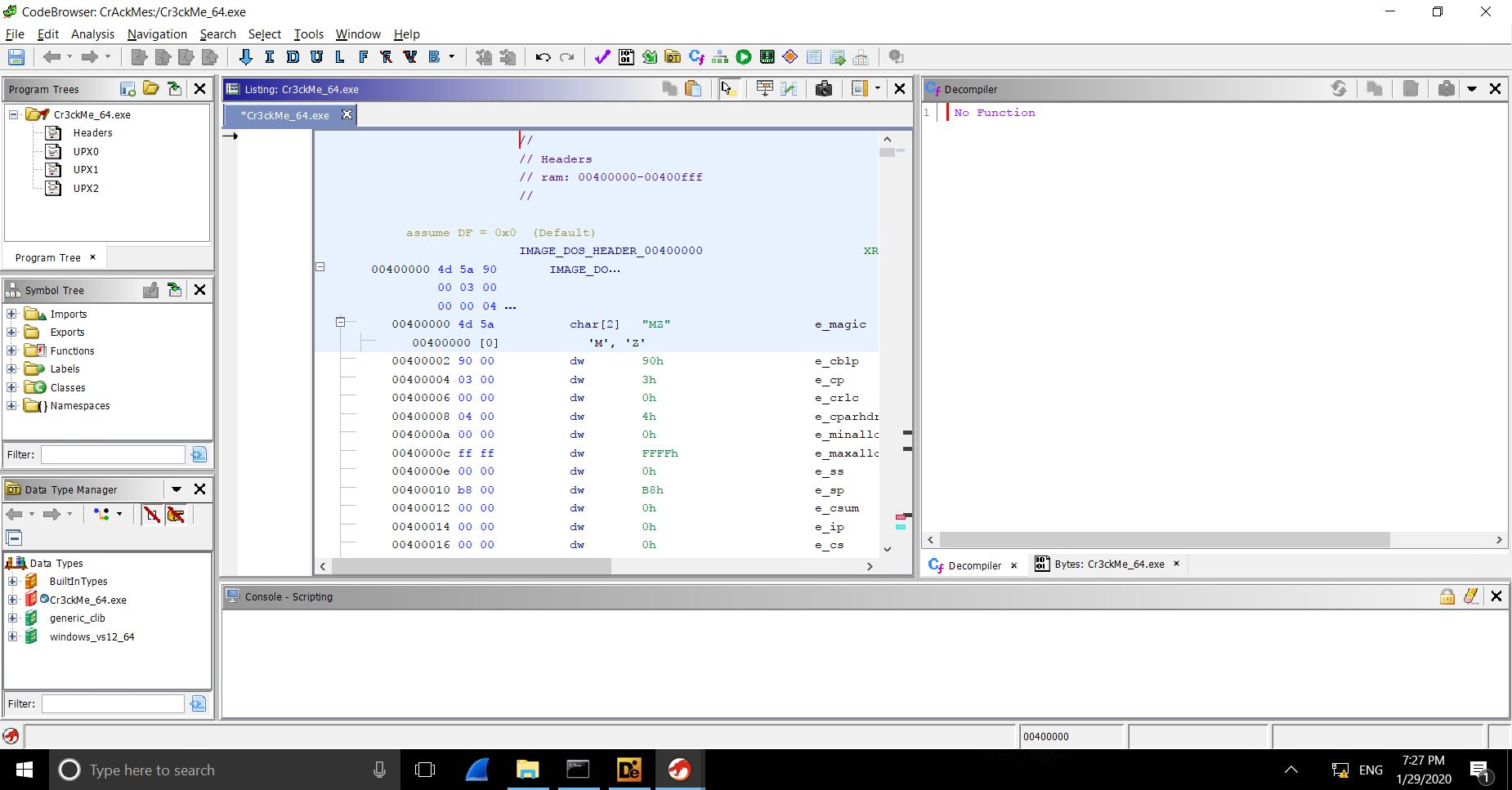


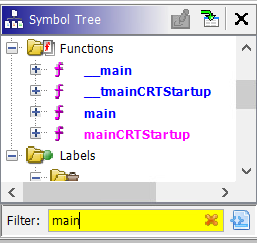
There are also other ways to check if a file is packed. We can look at the file with a hex editor such as HxD and spot UPX signatures such as the headers (UPX0, UPX1 etc.) and the UPX version (3.95).



Analysis & Unpacking

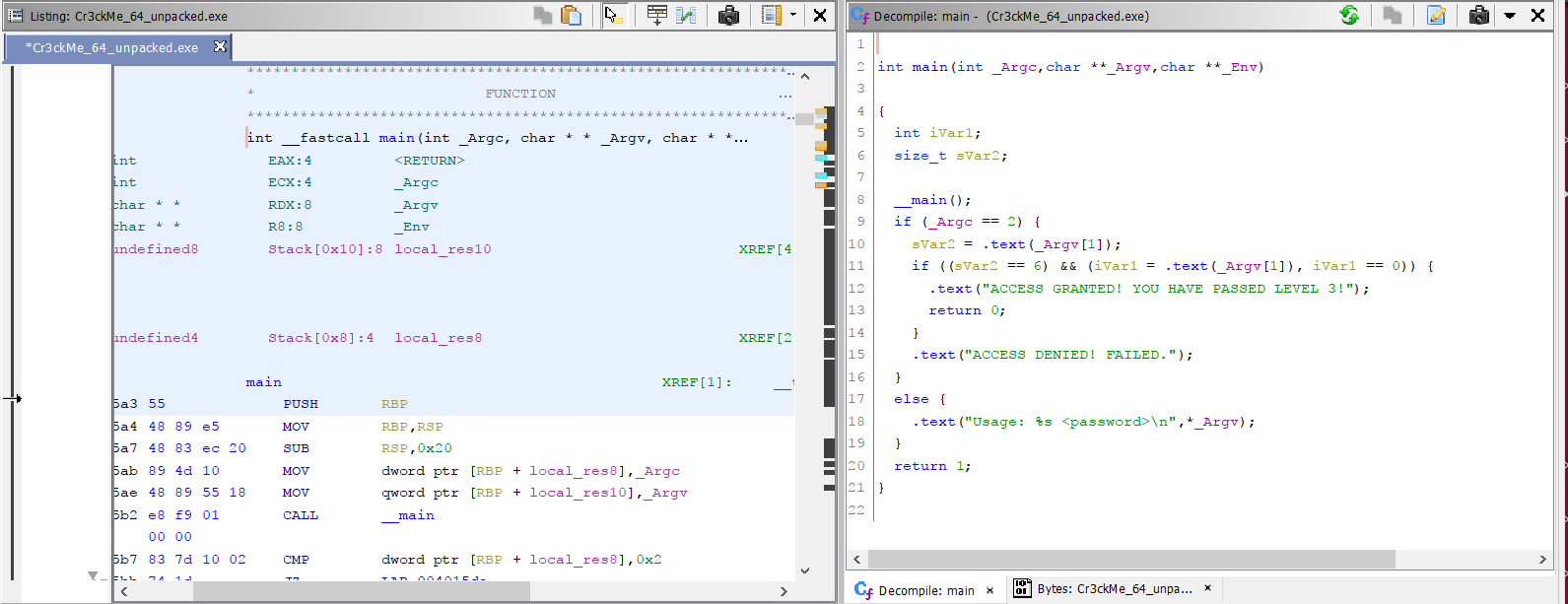
From the screenshot above, we can see that the executable has been packed by UPX, an open source packer. We can use UPX to unpack the executable as well, to retrieve the original unpacked file.

Now, if we open the unpacked executable in Ghidra, it should look just like Cr2ckMe.

Finding the main function

Using the Symbol Tree window, we can search for the main function of the program and navigate to that function by clicking on it.

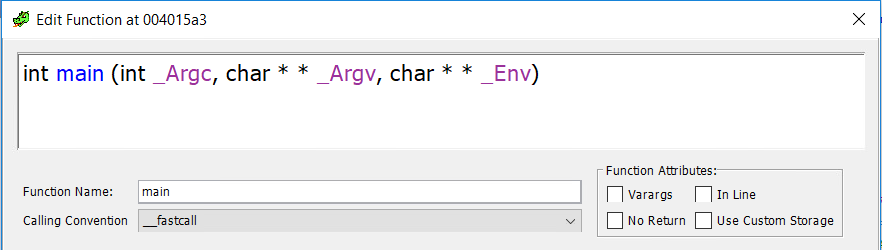
Working from the main function

From here, we will be focusing on the Listing window and the Decompile window.

Once the main function is found, the decompile window will show Ghidra’s decompilation of the Cr2ckMe. Looking at the first line in the decompile window:

int main(int \_Argc,char \*\*\_Argv,char \*\*\_Env)

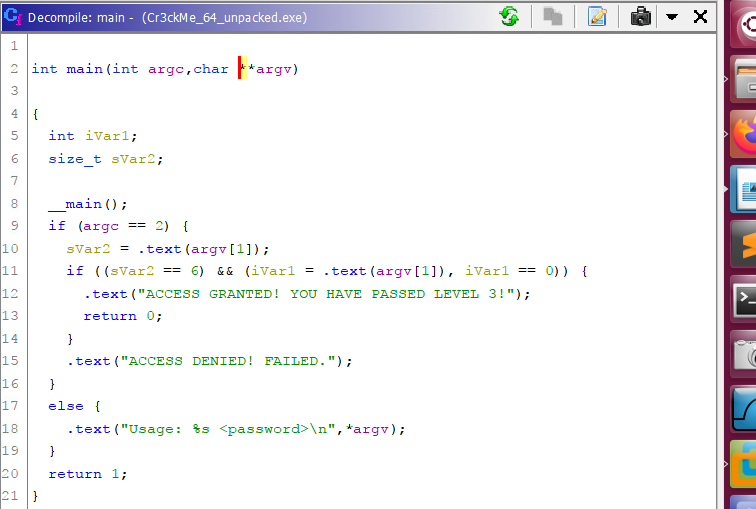
This looks exactly like a typical C program, other than the \_Env variable that is not used. We can make it cleaner by editing the function’s signature.  
(done by right clicking the first line and selecting “Edit function signature” from the menu)

From here, we simply input the C standard:

After doing so, our decompilation is cleaner with the only issue being the function names and variable names. However, looking at the code we can already identify the conditional statement that allows us to pass the Cr3ckMe.

int main(int argc,char\*\* argv)

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 = .text(argv[1]), iVar1 == 0))

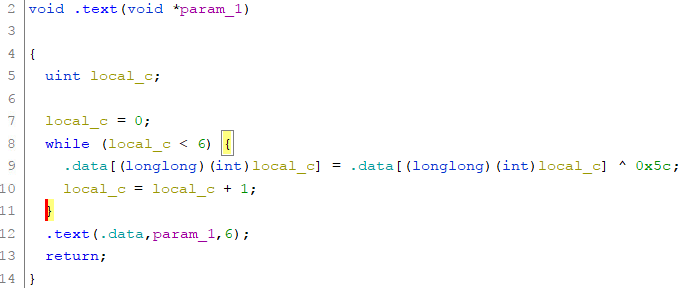
There are two factors that have to be True for this IF statement to be active:

* sVar2 == 6
* iVar1 = .text(argv[1]), iVar1 == 0

For the first factor, it tells us that the size of the password is 7 characters, since sVar2 is a size\_t type.

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

**.text #1 Function Breakdown**



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

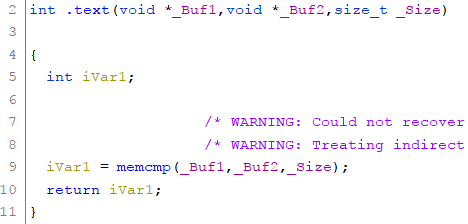
* a FOR loop (with local\_c variable as a counter) with -:
  + a .data variable (hard-coded variable)
  + index references to said variable
  + XOR function against 0x5C
* function that takes in:
  + a .data variable
  + additional parameter (password input)
  + size\_t value of 6

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

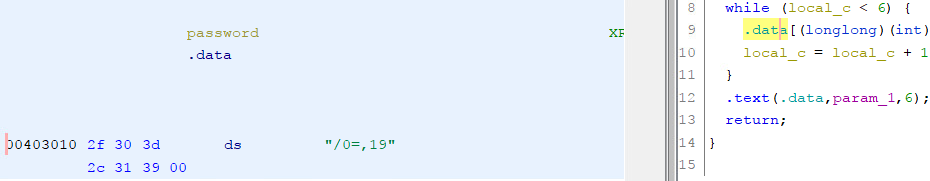
|  |
| --- |
| for(int counter = 0; counter < 6; counter++){  data[counter] = data[counter] ^ 0x5c;  }  function(data,input\_password,6); |

After simplifying the decompiled code, we can see that this function takes the hard-coded data variable and iterates through each byte and XOR’s it against 0x5C. Then, it gets put into another .text function. To look into this next function, we can simply double click the .text function.

**.text #2 Function Breakdown**

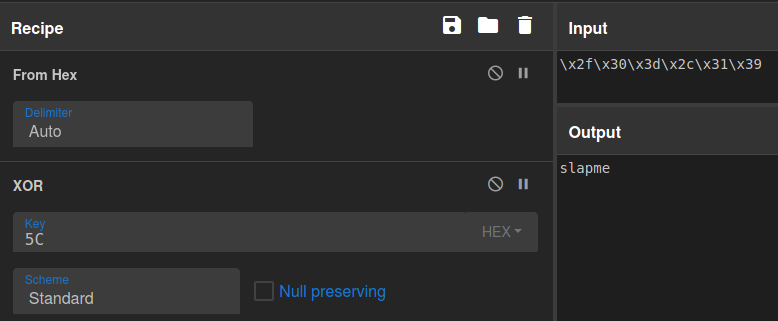
We can see that this function is just a memcmp function.

It takes in the XOR’d data string and the password input and compares them. If they match, it returns true.

**P****assword Found & Deciphered**

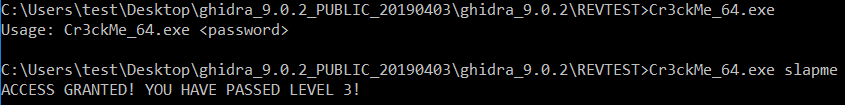
By double clicking on the .data variable, it brings us to the .data section of the executable with the value of the .data variable which is “/0=,19”. It looks like a garbled mess but this is our password.

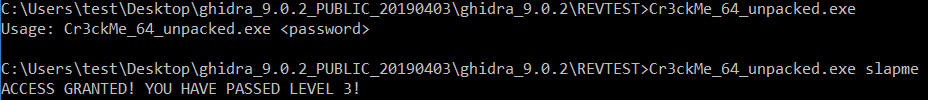
Since we know the way the program works, we know the password is stored before being XOR’d and compared.



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

As we can see, the password is slapme.

**Solved**  
This also works for the unpacked version, as packing only compresses and obfuscates the code without changing it.

**References**

[https://gchq.github.io/CyberChef/#recipe=From\_Hex('Auto')XOR(%7B'option':'Hex','string':'5C'%7D,'Standard',false)&input=XHgyY1x4MzVceDMyXHgzZlx4MzRceDMxXHgzOQ](https://gchq.github.io/CyberChef/" \l "recipe=From_Hex('Auto')XOR({'option':'Hex','string':'5C'},'Standard',false)&input=XHgyY1x4MzVceDMyXHgzZlx4MzRceDMxXHgzOQ) (XOR)