[REV] crackme 🩸 | rydzze

📚 Description

Lets brush up your **rusty** RE skill. Password for zip **umcs** ... Author: ayam

🤔 Hints

find xrefs to **messagebox api**. differentiate **goodboy/badboy**

✨ Walkthrough

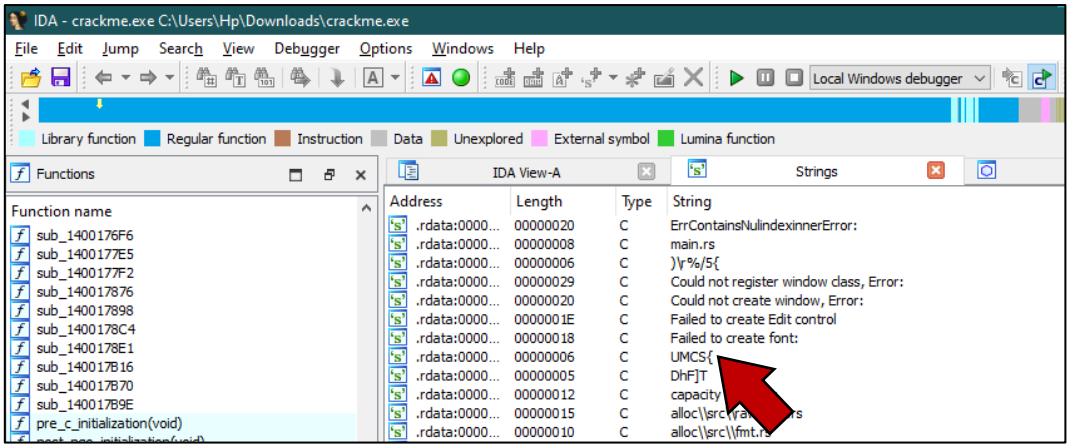
YIPPEE FIRST BLOOD 🩸 ... Reverse engineering challenge written in **Rust**, how bad/hard/difficult could it be right :D? *huhuhu* anyways, this is my write-up for it.

```
rydzze /mnt/c/Users/Hp/Downloads ❯ file crackme.exe
crackme.exe: PE32+ executable for MS Windows 6.00 (GUI), x86-64, 3 sections

rydzze /mnt/c/Users/Hp/Downloads ❯ strings crack*
!This program cannot be run in DOS mode.
_t@I_w8
_Rich}8
UPX0
UPX1
UPX2
```

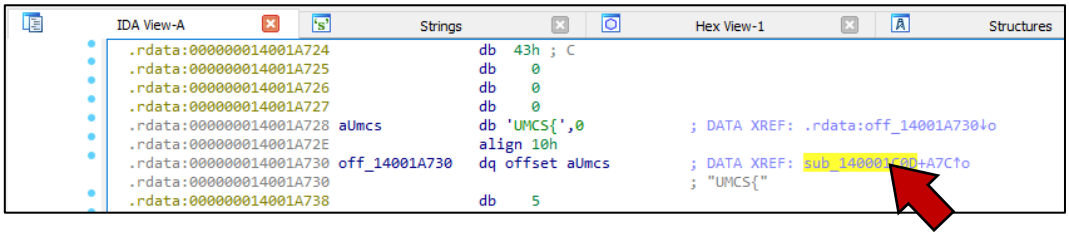
In short, we are given an **EXE binary** which **utilised the WinAPI** to make the GUI. Since this is rev chal, of course we will use `strings` command on the binary so that's what I did, and we found out that there are **UPX** in the output which means that this binary has been **compressed using UPX**.

```
rydzze /mnt/c/Users/Hp/Downloads ❯ upx -d crackme.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2025
UPX 5.0.1       Markus Oberhumer, Laszlo Molnar & John Reiser    May 6th 2025

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    138240 <-     68096   49.26%    win64/pe     crackme.exe

Unpacked 1 file.

rydzze /mnt/c/Users/Hp/Downloads ❯ file crackme.exe
crackme.exe: PE32+ executable for MS Windows 6.00 (GUI), x86-64, 5 sections
```

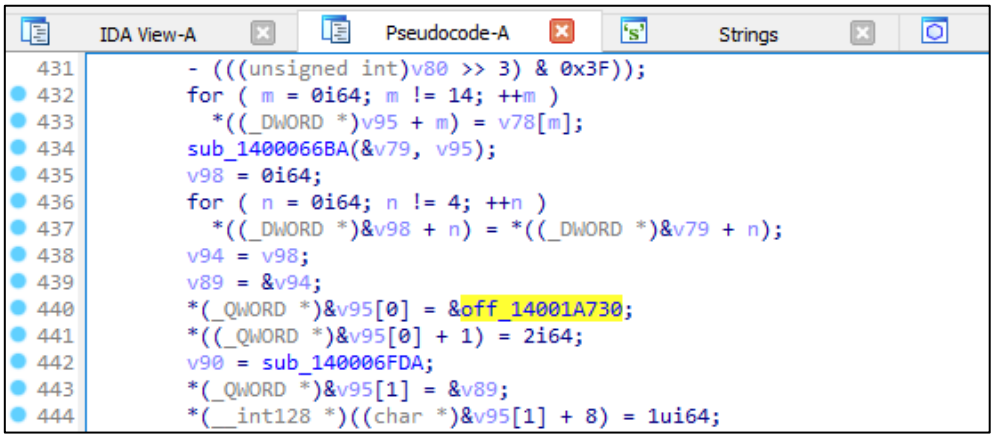
Now, let's **use UPX** to **decompress the binary** so that we can disassemble and debug it to understand what is going on.



In this case, I'm using IDA so that we can do both static and dynamic analyses. *As usual*, remember to **check the strings** subview okayy XD. Once we went in, we found the flag format, `UMCS{` inside and then just double-click on it.



Now, we know that **this string** was **used in a function** based on the DATA_XREF and again, *just double-click on it* and press F5 to view the pseudocode ...

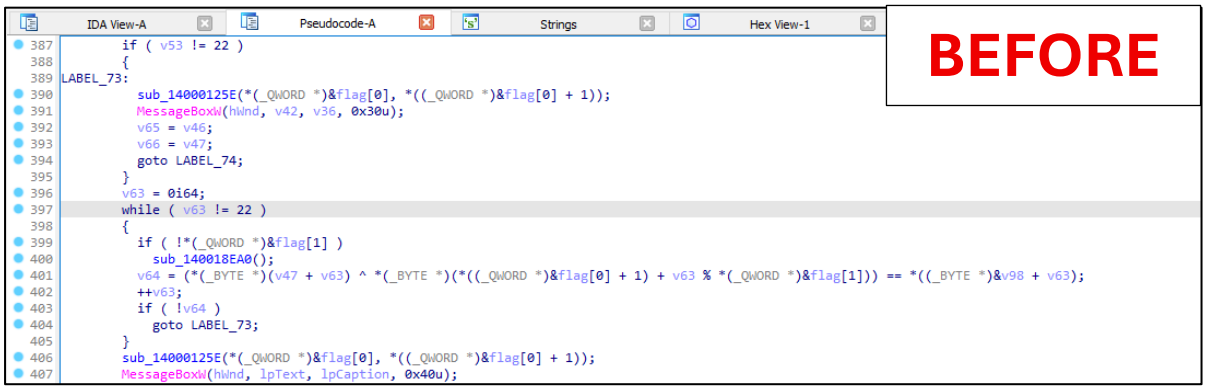


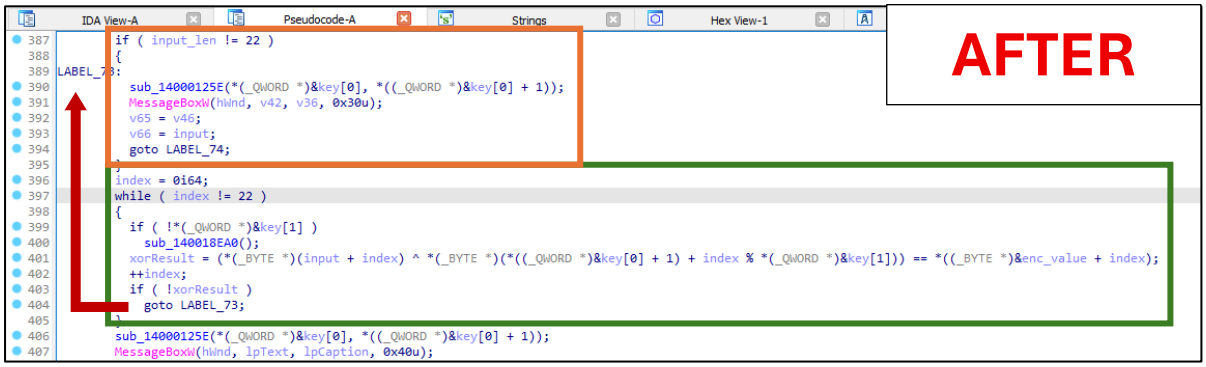
Yeah, I know, there is lot of stuff going on in this function but don't worry about it lol.

Based on the above snippet, we can see that **v95 points to** the `UMCS{` string which means that we are in the right path lmao, *theres definitely something interesting right here*.

After that, I traced all of the **MessageBox API** called in the function by **placing breakpoint** to each one of them (*not really*) and then perform dynamic analysis lol, hoping to be stopped and later on stumble across *valuable* information.

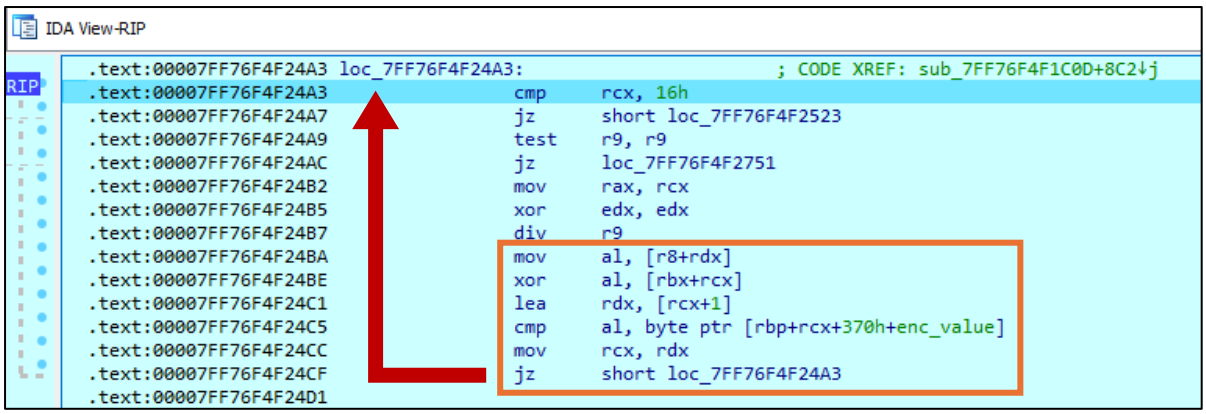
Analysis DONE … Without further ado, let's quickly understand the processes.





I've renamed some of the variables to make things *easier*. Firstly, at the top we found an **if-statement** that will **check the length** of our input. If the input_len is not equal to 22, it will display the 'Try again' MessageBox … That said, now we know that the **input should be 22 characters** in total.

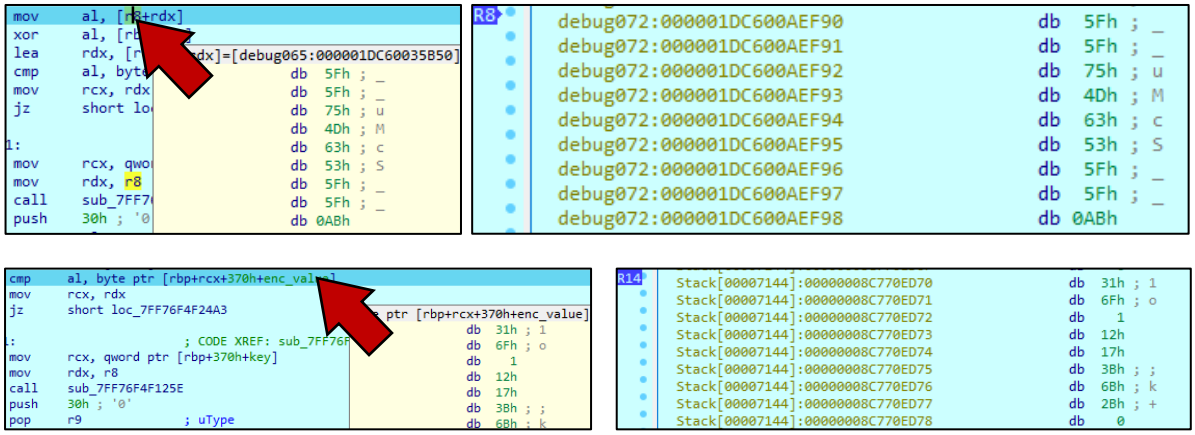
Next, there is a **while-loop** that will be **running for 22 times**, performing **XOR** on each of the input characters with a *key* and then **comparing the result** with some values. If the XOR result is 1 which means the result is not the same, it will display the 'Try again' MessageBox. *Well, let's dive a little bit deeper, shall we?* Insert **breakpoint at line 401** and **run the debugger**.



I will be explaining what is happening inside the highlighted box, you may try it out by yourself and then **monitor the registers** and **flags** :D. In short, it will;

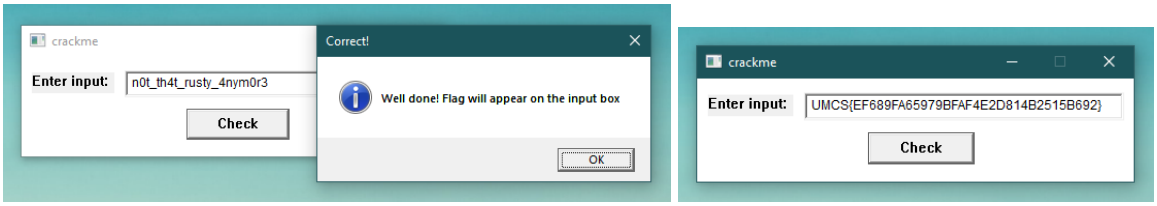
1. Move a byte (*character*) from the key to the AL register, low 8 bits.
2. XOR the AL register with our input.
3. Increase the RDX (data) register by 1.
4. Compare XOR result in AL register with the expected value. **ZF TRIGGERED!?**
5. Move the value in RDX register into RCX (count) register (*for count, index, etc.*).
6. Jump to the beginning of the loop if ZF is 1 (*our XOR result is okayyy, same value*).

Alright now, **obtain the value** for both **enc_value** and **key**. However, I can't obtain the value directly during static analysis (*let me know if you know da wae, sifu*) ... Knowing what kind of person I am, I just *copy paste* the value during dynamic analysis lol.



Last but not least, this is the **script** to **reverse the process** and **get the expected input**.

```
key = '__uMcS__'
enc_value = [0x31, 0x6F, 0x1, 0x12, 0x17, 0x3B, 0x6B, 0x2B, 0x0, ...]

input = ''.join(chr(enc_value[i] ^ ord(key[i % len(key)])) for i in range(len(enc_value)))
print(input[:22])

# n0t_th4t_rusty_4nym0r3
```



Flag > `UMCS{EF689FA65979BFAF4E2D814B2515B692}`

[DEFENSE] babysc_note | rydzze

Description

you guys had fun asking chatgpt for the first babysc challenge?? goodluck with this one

Author: Capang

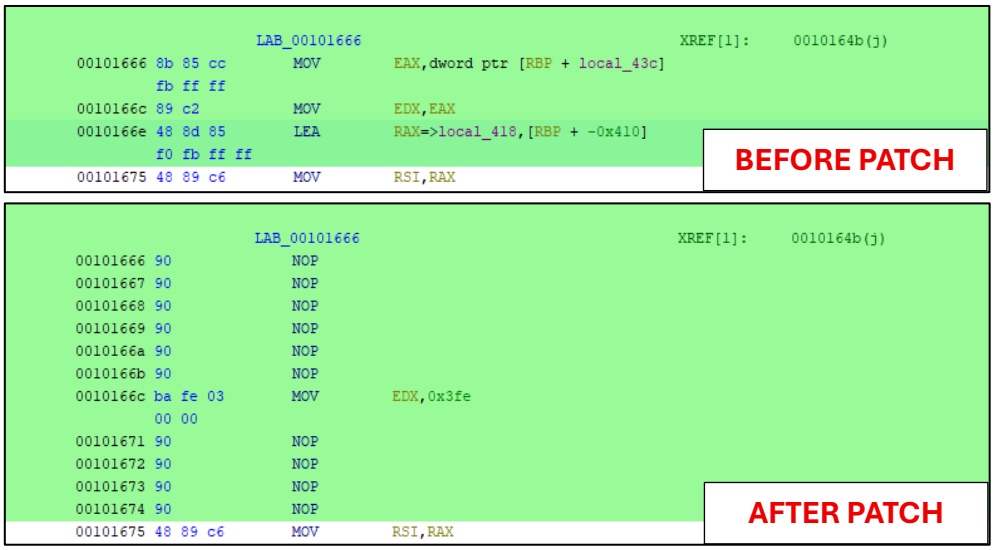
Walkthrough

Due to *unbearable skill issues*, I didn't managed to exploit it lol *sobsob* ...

"*you guys had fun asking chatgpt for the first babysc challenge??*" yeah and that is the reason why I'm going to use it once again to patch the binary HAHAHAHAH.

```
read(0,&local_418,(ulong)local_43c);
```

"*The vulnerability is a stack-based buffer overflow:* ***read() allows up to 1023 bytes*** *to be* ***written into local_418, which is only 8 bytes****, overwriting adjacent stack data. This can lead to* ***arbitrary code execution****.*" – well said, ChatGPT. **Patch the opcode** with Ghidra.



After the patch, the **user-controlled size is replaced** with a **fixed value 0x3fe**, preventing buffer overflow by limiting how much data can be read.

Thank you :)