

# [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**.

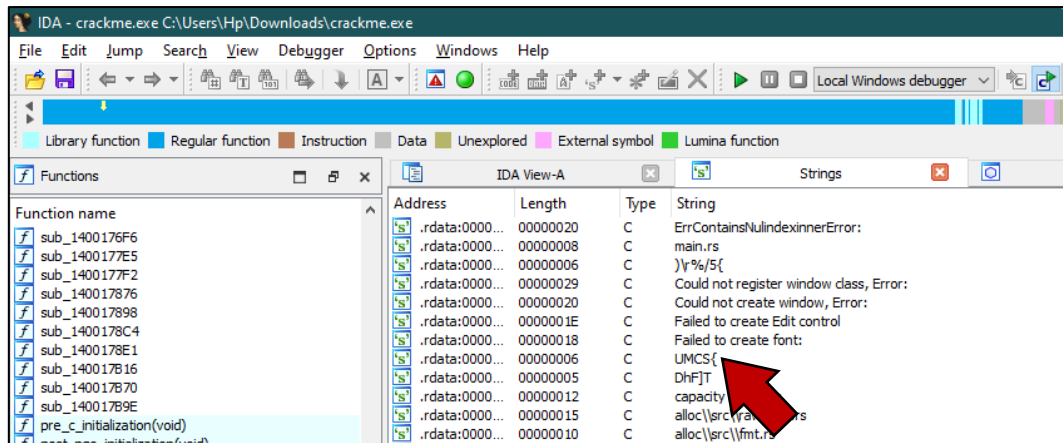
```
rydzze /mnt/c/Users/Hp/Downloads > upx -d crackme.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2025
UPX 5.0.1      Markus Oberhumer, Laszlo Molnar & John Reiser    May 6th 2025

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

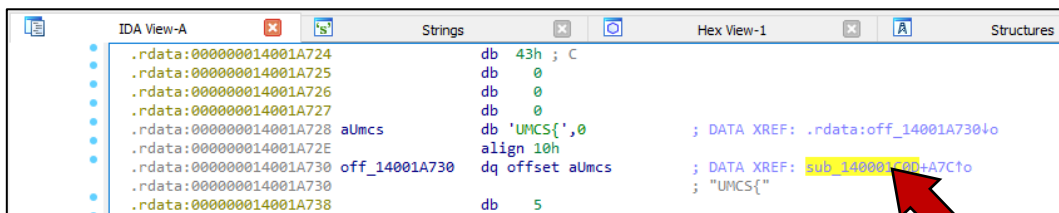
Unpacked 1 file.

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

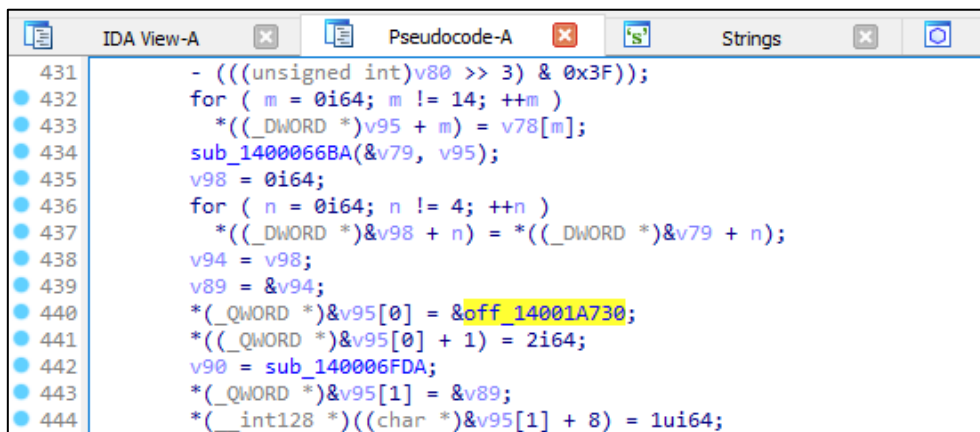
Now, let's **use UPX to decompress the binary** so that we can disassemble and debug it to understand what is going on.



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 okay 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.

IDA View-A

Pseudocode-A

Strings

Hex View-1

```
387 if ( v53 != 22 )
388 {
389 LABEL_73:
390 sub_14000125E((__QWORD *)&flag[0], *((_QWORD *)&flag[0] + 1));
391 MessageBoxW(hwnd, v42, v36, 0x30u);
392 v65 = v46;
393 v66 = v47;
394 goto LABEL_74;
395 }
396 v63 = 0i64;
397 while ( v63 != 22 )
398 {
399 if ( !((__QWORD *)&flag[1] )
400 sub_140018EA0();
401 v64 = *((_BYTE *) (v47 + v63) ^ *((_BYTE *) ((__QWORD *)&flag[0] + 1) + v63 % ((__QWORD *)&flag[1])) == *((_BYTE *)&v98 + v63);
402 ++v63;
403 if ( !v64 )
404 goto LABEL_73;
405 }
406 sub_14000125E((__QWORD *)&flag[0], *((_QWORD *)&flag[0] + 1));
407 MessageBoxW(hwnd, lpText, lpCaption, 0x40u);
```

BEFORE

IDA View-A

Pseudocode-A

Strings

Hex View-1

```
387 if ( input_len != 22 )
388 {
389 LABEL_73:
390 sub_14000125E((__QWORD *)&key[0], *((_QWORD *)&key[0] + 1));
391 MessageBoxW(hwnd, v42, v36, 0x30u);
392 v65 = v46;
393 v66 = input;
394 goto LABEL_74;
395 }
396 index = 0i64;
397 while ( index != 22 )
398 {
399 if ( !((__QWORD *)&key[1] )
400 sub_140018EA0();
401 xorResult = *((_BYTE *) (input + index) ^ *((_BYTE *) ((__QWORD *)&key[0] + 1) + index % ((__QWORD *)&key[1])) == *((_BYTE *)&enc_value + index);
402 ++index;
403 if ( !xorResult )
404 goto LABEL_73;
405 }
406 sub_14000125E((__QWORD *)&key[0], *((_QWORD *)&key[0] + 1));
407 MessageBoxW(hwnd, lpText, lpCaption, 0x40u);
```

AFTER

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**.

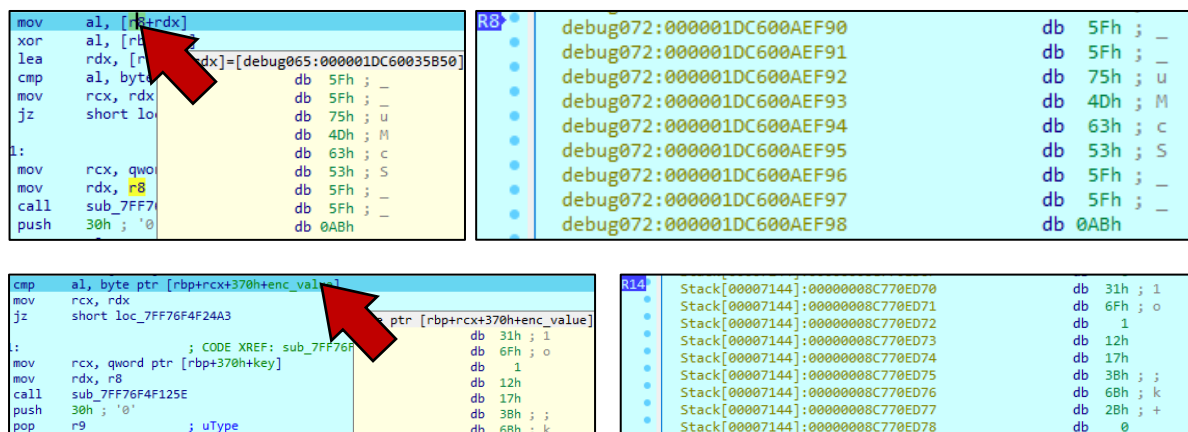
IDA View-RIP

```
.text:00007FF76F4F24A3 loc_7FF76F4F24A3: ; CODE XREF: sub_7FF76F4F1C0D+8C24j
.text:00007FF76F4F24A3 cmp rcx, 16h
.text:00007FF76F4F24A7 jz short loc_7FF76F4F2523
.text:00007FF76F4F24A9 test r9, r9
.text:00007FF76F4F24AC jz loc_7FF76F4F2751
.text:00007FF76F4F24B2 mov rax, rcx
.text:00007FF76F4F24B5 xor edx, edx
.text:00007FF76F4F24B7 div r9
.text:00007FF76F4F24BA mov al, [r8+rdx]
.text:00007FF76F4F24BE xor al, [rbx+rcx]
.text:00007FF76F4F24C1 lea rdx, [rcx+1]
.text:00007FF76F4F24C5 cmp al, byte ptr [rbp+rcx+370h+enc_value]
.text:00007FF76F4F24CC mov rcx, rdx
.text:00007FF76F4F24CF jz short loc_7FF76F4F24A3
.text:00007FF76F4F24D1
```

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 okayyyy, 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.



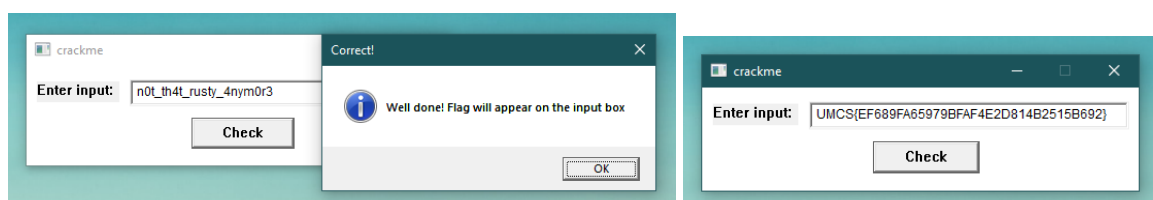
The screenshot shows a debugger interface with three panels. The left panel displays assembly code with a red arrow pointing to the instruction `mov al, [rdi+rdx]`. The middle panel shows the state of the R8 register, with a red arrow pointing to the instruction `cmp al, byte ptr [rbp+rcx+370h+enc_value]`. The right panel shows the state of the R14 register, displaying a list of memory addresses and their corresponding values.

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 | rydze

## 📖 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 HAHAAHAHAH.

```
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.

LAB_00101666				XREF[1]:	0010164b(j)
00101666	8b 85 cc	MOV	EAX,dword ptr [RBP + local_43c]		
	fb ff ff				
0010166c	89 c2	MOV	EDX,EAX		
0010166e	48 8d 85	LEA	RAX=>local_418,[RBP + -0x410]		
	f0 fb ff ff				
00101675	48 89 c6	MOV	RSI,RAX		
				BEFORE PATCH	
LAB_00101666				XREF[1]:	0010164b(j)
00101666	90	NOP			
00101667	90	NOP			
00101668	90	NOP			
00101669	90	NOP			
0010166a	90	NOP			
0010166b	90	NOP			
0010166c	ba fe 03	MOV	EDX,0x3fe		
	00 00				
00101671	90	NOP			
00101672	90	NOP			
00101673	90	NOP			
00101674	90	NOP			
00101675	48 89 c6	MOV	RSI,RAX		
				AFTER PATCH	

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 :)