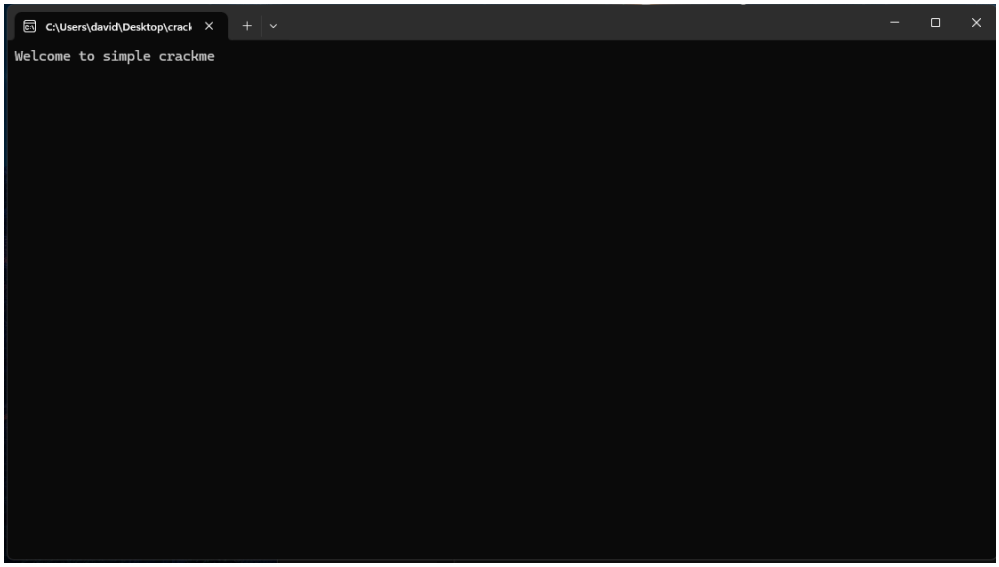
Crackme: https://crackmes.one/crackme/6929c3af2d267f28f69b8196

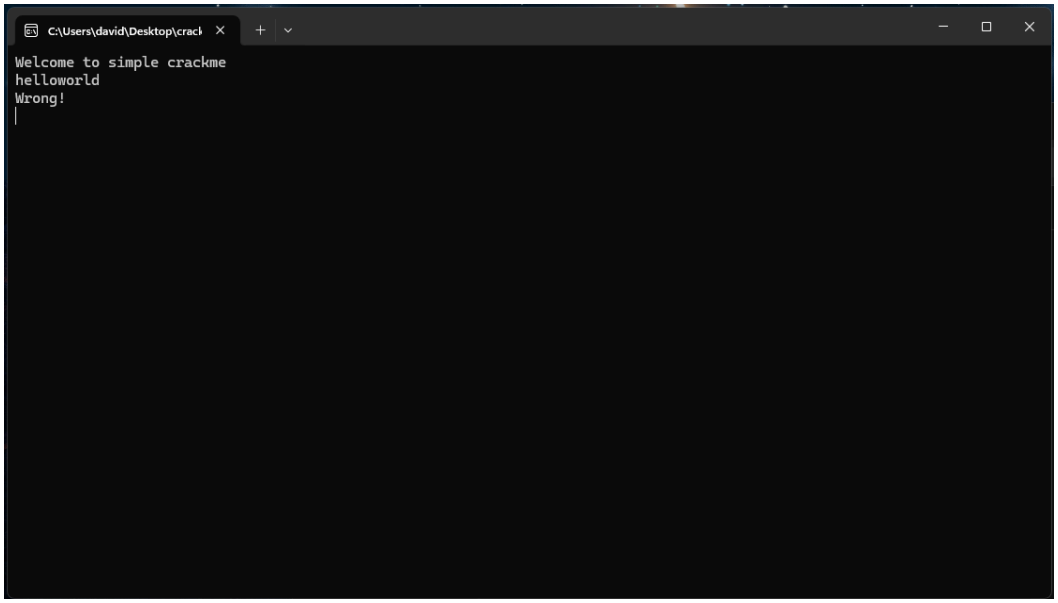
Author: illusionxxx Software Used: x64dbg Writeup By: SenorGPT

Platform Windows	Difficulty: 1.5	Quality: 4.0	Arch: x86-64	Language: C/C++

Regular execution (startup):



Wrong answer provided - proceeded by termination:

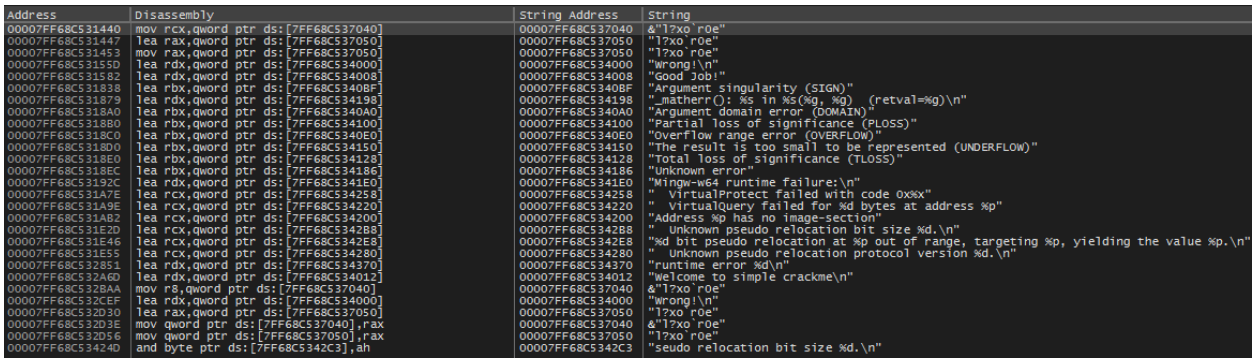


Initial thoughts on approach:

- Search for string references of given strings: “Welcome to simple crackme” and “Wrong!”, and other string references picked up by x64dbg.
- Set breakpoints onto process termination / exit.
- Set breakpoints onto console input API methods.

String References Approach:

String reference list:



Address	Disassembly	String Address	String
00007FF68C31440D	mov rcx,qword ptr ds:[7FF68C537040]	00007FF68C537040	&"l?xo`r0e"
00007FF68C31447	lea rax,qword ptr ds:[7FF68C537050]	00007FF68C537050	"l?xo`r0e"
00007FF68C31453	mov rax,qword ptr ds:[7FF68C537050]	00007FF68C537050	"l?xo`r0e"
00007FF68C3155D	lea rdx,qword ptr ds:[7FF68C534000]	00007FF68C534000	"Wrong!\n"
00007FF68C31582	lea rdx,qword ptr ds:[7FF68C534008]	00007FF68C534008	"Good Job!"
00007FF68C31838	lea rbx,qword ptr ds:[7FF68C5340BF]	00007FF68C5340BF	"Argument singularity (SIGN)"
00007FF68C31879	lea rdx,qword ptr ds:[7FF68C534198]	00007FF68C534198	"_matherr(): %s in %s(%g, %g) (retval=%g)\n"
00007FF68C318A0	lea rbx,qword ptr ds:[7FF68C5340A0]	00007FF68C5340A0	"Argument domain error (DOMAIN)"
00007FF68C318B0	lea rbx,qword ptr ds:[7FF68C534100]	00007FF68C534100	"Partial loss of significance (PLOSS)"
00007FF68C318C0	lea rbx,qword ptr ds:[7FF68C5340E0]	00007FF68C5340E0	"Overflow range error (OVERFLOW)"
00007FF68C318D0	lea rbx,qword ptr ds:[7FF68C534150]	00007FF68C534150	"The result is too small to be represented (UNDERFLOW)"
00007FF68C318E0	lea rbx,qword ptr ds:[7FF68C534128]	00007FF68C534128	"Total loss of significance (TLOSS)"
00007FF68C318EC	lea rbx,qword ptr ds:[7FF68C534186]	00007FF68C534186	"Unknown error"
00007FF68C3192C	lea rdx,qword ptr ds:[7FF68C5341E0]	00007FF68C5341E0	"Mingw-w64 runtime failure:\n"
00007FF68C31A7E	lea rcx,qword ptr ds:[7FF68C534258]	00007FF68C534258	" VirtualProtect failed with code 0x%x"
00007FF68C31A9E	lea rcx,qword ptr ds:[7FF68C534220]	00007FF68C534220	" VirtualQuery failed for %d bytes at address %p"
00007FF68C31AB2	lea rcx,qword ptr ds:[7FF68C534200]	00007FF68C534200	"Address %p has no image-section"
00007FF68C31E2D	lea rcx,qword ptr ds:[7FF68C5342B8]	00007FF68C5342B8	" Unknown pseudo relocation bit size %d.\n"
00007FF68C31E46	lea rcx,qword ptr ds:[7FF68C5342E8]	00007FF68C5342E8	"%d bit pseudo relocation at %p out of range, targeting %p, yielding the value %p.\n"
00007FF68C31E55	lea rcx,qword ptr ds:[7FF68C534280]	00007FF68C534280	" Unknown pseudo relocation protocol version %d.\n"
00007FF68C32851	lea rdx,qword ptr ds:[7FF68C534370]	00007FF68C534370	"runtime error %d\n"
00007FF68C32A6D	lea rdx,qword ptr ds:[7FF68C534012]	00007FF68C534012	"Welcome to simple crackme\n"
00007FF68C32BAA	mov r8,qword ptr ds:[7FF68C537040]	00007FF68C537040	&"l?xo`r0e"
00007FF68C32CEF	lea rdx,qword ptr ds:[7FF68C534000]	00007FF68C534000	"Wrong!\n"
00007FF68C32D30	lea rax,qword ptr ds:[7FF68C537050]	00007FF68C537050	"l?xo`r0e"
00007FF68C32D3E	mov qword ptr ds:[7FF68C537040],rax	00007FF68C537040	&"l?xo`r0e"
00007FF68C32D56	mov qword ptr ds:[7FF68C537050],rax	00007FF68C537050	"l?xo`r0e"
00007FF68C3424D	and byte ptr ds:[7FF68C5342C3],ah	00007FF68C5342C3	"seudo relocation bit size %d.\n"

At first glance, the string "l?xo`r0e" seems smelly. So, I threw it in as an answer. No bueno. It is more than likely that it is an encoded or transformed version of the actual correct flag/key.
I added breakpoints around where the string references “Wrong!” had led me in the disassembly view.
Stepping through the main program logic, I see my input of “helloworld” get passed to an encoding function of sorts which transforms the data into “boffe}exfn”.

This converted string suspiciously looks like a Caesar cipher, or a shift cipher. It seemed like a Caesar cipher since each character appeared to map to the same character (l to f, o to e, etc). Using a simple Caser cipher decoder, I tried going through shifts but none of them seemed to match up. Womp, womp, womp.

I knew I was on the right track though, and before attempting to reverse engineer the actual encoding function I had two more thoughts.

1. I noticed that one of the registers in the main comparison code loop was 0x0A (10) which led me down the path that it might be a per-byte XOR with a fixed key.
2. Execute a cheeky route by just putting all the characters into a string and passing it through the input to see what it converts it to, which then we can create a lookup table for the encoded password we already found.

I will save option 2) if I'm wrong about option 1) and get nowhere, and if I have no other leads.

Snippet of assembly code calling encoding function:

```
mov qword ptr ss:[rsp+78],rax      [rsp+78] = strlen
call a.7FF68C531470                call encoding function
mov rdx,qword ptr ss:[rsp+50]      [rsp+50] is somehow also the strlen?
```

Let's try to reverse the per-byte XOR with a fixed key for "boffe}exfn" back to "helloworld".

"helloworld" ^ 0x0A = "boffe}exfn"

h = 0x68	0x68 ^ 0x0A = 0x62	b
e = 0x65	0x65 ^ 0x0A = 0x6F	o
l = 0x6C	0x6C ^ 0x0A = 0x66	f
l = 0x6C	0x6C ^ 0x0A = 0x66	f
o = 0x6F	0x6F ^ 0x0A = 0x65	e
w = 0x77	0x77 ^ 0x0A = 0x7D	}
o = 0x6F	0x6F ^ 0x0A = 0x65	e
r = 0x72	0x72 ^ 0x0A = 0x78	x
l = 0x6C	0x6C ^ 0x0A = 0x66	f
d = 0x64	0x64 ^ 0x0A = 0x6E	n

Perfect, it's a match!

That means that it is indeed doing a per-byte XOR with a fixed key on the user input and then comparing it to the string “l?xo`r0e”. Which means that if we just reverse the cipher on that string we will have the flag/key. Since XOR is symmetric, we can XOR “l?xo`r0e” with “0x0A” to get the decoded flag/key.

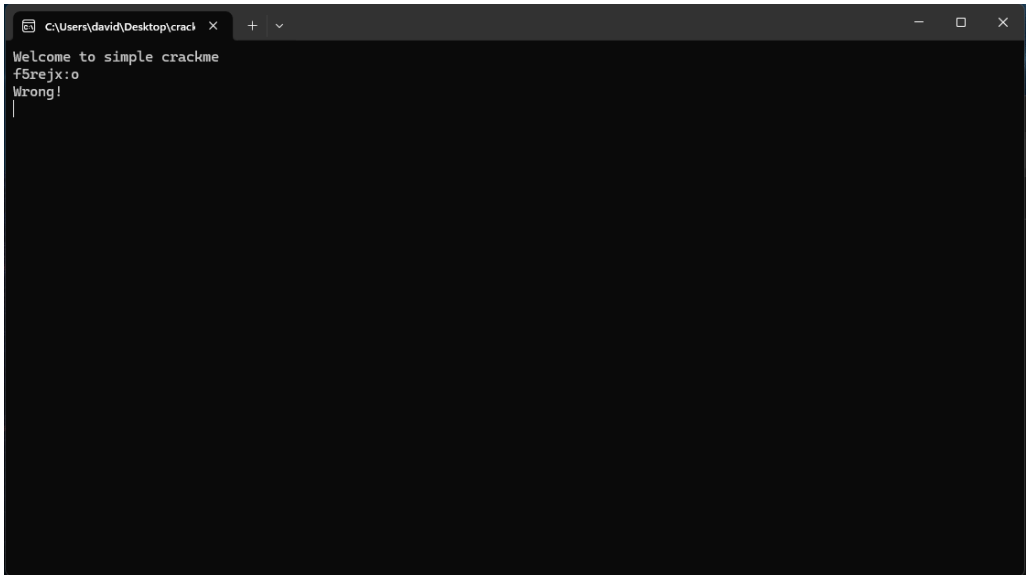
So let’s XOR that string, get the supposed flag/key, and plug it into the CTF to see what happens.

"l?xo`r0e" ^ 0x0A = "f5rejx:o"

l = 0x6C	0x6C ^ 0x0A = 0x66	f
? = 0x3F	0x3F ^ 0x0A = 0x35	5
x = 0x78	0x78 ^ 0x0A = 0x72	r
o = 0x6F	0x6F ^ 0x0A = 0x65	e
` = 0x60	0x60 ^ 0x0A = 0x6A	j
r = 0x72	0x72 ^ 0x0A = 0x78	x
0 = 0x30	0x30 ^ 0x0A = 0x3A	:
e = 0x65	0x65 ^ 0x0A = 0x6F	o

Final flag/key from per-byte XOR with a fixed key: *“f5rejx:o”*

Snickers.



It seems that the result after the encoding function ran for *“f5rejx:o”* is *“n=zmbp2g”*, not *"l?xo`r0e"...*

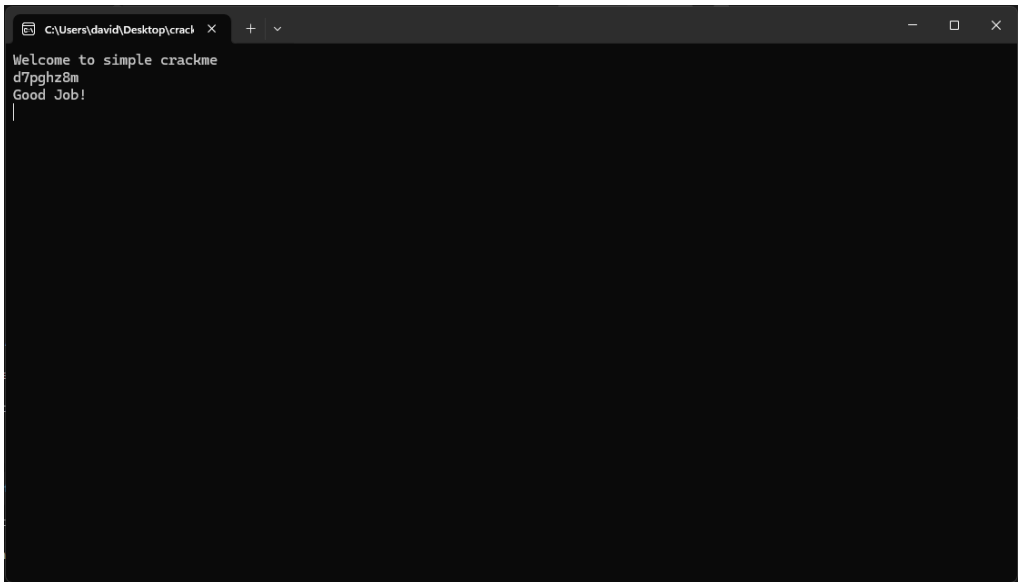
Time for some sanity checks. Running the program with “boffe}exfn” I see that the result after the encoding function runs is “helloworld”, which is confusing... It seems that I am on the right path but making a mistake somewhere or not seeing something.
After some digging around, it seems that the XOR key changes depending on the length of the string. In fact, it actually uses the length of the string itself as the XOR key.
So, where I went wrong with my previous XOR on the encoded flag/key is that I used the wrong XOR key. The real XOR key is based on the length of the actual input provided by the user. Let's fix that.

"l?xo`r0e" ^ 0x08 = "d7pghz8m"

l = 0x6C	0x6C ^ 0x08 = 0x64	d
? = 0x3F	0x3F ^ 0x08 = 0x37	7
x = 0x78	0x78 ^ 0x08 = 0x70	p
o = 0x6F	0x6F ^ 0x08 = 0x67	g
` = 0x60	0x60 ^ 0x08 = 0x68	h
r = 0x72	0x72 ^ 0x08 = 0x7A	z
0 = 0x30	0x30 ^ 0x08 = 0x38	8
e = 0x65	0x65 ^ 0x08 = 0x6D	m

NEW final flag/key from per-byte XOR with a fixed key: *“d7pghz8m”*.

Okay, let’s plug this puppy in and see what happens.



Ohh yeah!! Time to go celebrate with a nice ice cold refreshing beverage. Maybe a Nestea.

Final Thoughts:

- This was my first ever CTF. I started learning reverse engineering and more low level operations about a few months ago. Therefore, this is also my first writeup solution. If I missed anything or completely missed the ball with this writeup, please don't be shy to offer constructive criticism as that is the only way I can learn, grow, and improve the quality of output.
- I never actually stepped into the encoding function as there were plenty of leads for me to lean into prior to going that deep.
- Another aforementioned cheeky solution would have been to supply the input with a string of all characters, then building a lookup table to decode the found encrypted flag/key. I did find a check within the assembly that seemed to validate the user input length to a length of 15, but it seemed to continue to progress down the encoding code path regardless.
- Even though this is a "simple" crackme, it did take some time for me to figure it out. Which has given me even more respect to those that have mastered this art. Nevertheless, I am still proud of this achievement 🙂.
- More detailed explanation of what led me to believe it was a per-byte XOR with a fixed key after my Caesar cipher solution failed. When stepping through, I noticed one of the registers that was used in the main character comparison loop was 0x0A. At first, I didn't think anything of this until the Caesar cipher lead went cold.

  Main character comparison code loop:

  I then took a closer look at the encoded value of "boffe}exfn" from "helloworld" and noticed that the character transformations had different offsets. For example, take the first two characters "he" from "helloworld":
  h (0x68) — b (0x62) = -0x06 (0x62 - 0x68)
  e (0x6F) — o (0x65) = +0x0A (0x6F - 0x65)

  So I then proceeded to XOR the two values together which gave me a constant: 0x0A.
  h (0x68) ^ b (0x62) = 0x0A
  e (0x6F) ^ o (0x65) = 0x0A
  l (0x6C) ^ f (0x66) = 0x0A

This is why at first I thought that it performed a XOR on each character with the key of 0x0A, but as I came to discover this was only true for strings with the length of 10. Which is why my first XOR cipher guess at decoding the found encrypted key failed, because the XOR key was supposed to be 0x08 instead of 0x0A.