Greetings in this article I will RE this crackme. The rules are to find how the serial key is generated (Not sure honestly, heh, cus they weren't strictly defined in the author's page, so yeah...).
Let's get started

0.1 Tools required

- x64dbg
- ghidra/rizin/binja/ida
- frida
- a lot of free time
- luck, attention & some deduction skills

1 Finding clues

1.1 input

On this stage I tried to locate what comparison rules are and is there way to fool them. The following asm listings are somehow related, to this process, i will comment them if there will be something significant.

```
mov qword ptr ss:[rsp+8],rbx            ; enter main code
mov qword ptr ss:[rsp+10],rsi
mov qword ptr ss:[rsp+18],rdi
push rbp
lea rbp,qword ptr ss:[rsp-20]
sub rsp,120
mov rax,qword ptr ds:[7FF77AAA61A0]
xor rax,rsp
mov qword ptr ss:[rbp+10],rax
lea rcx,qword ptr ss:[rbp-30]           ; 4251417266796500, this
    value is preserved between launches
call crackme.7FF77AA340A0               ; looks useless
nop
lea rdx,qword ptr ds:[7FF77AAAC588]     ; KeyfrAQBc8Wsa string
    appeared at this point. Note: this value appears before main
    function
lea rcx,qword ptr ss:[rbp-78]
call crackme.7FF77AA35F40               ; copies some value into
    xmm and jumps out
mov r8,rax                              ; some string appears here
    that looks like part of crypto system
lea rdx,qword ptr ss:[rbp-30]
lea rcx,qword ptr ss:[rbp-58]
call crackme.7FF77AA343F0               ; this function returns
    pGeneratedSerial
```

Listing 1: Begining of main function

Among interesting: note that from that point we can see from which location our serial key is coming from. We will return to 19-th line later after we investigate input handling. Let's take a look at decompiler output:

```
1  FUN_1400040a0(local_58);
2  pauVar5 = (undefined (*) [32])FUN_140005f40(local_a0,(undefined
     (*) [32])&DAT_14007c588);
3  FUN_1400043f0((undefined8 *)&local_80,local_58,pauVar5);
4  local_b0 = (char *)0x0;
5  local_a8 = 0xf;
6  local_c0[0] = (undefined8 ****)0x0;
7  FUN_140006640((undefined (*) [32])local_c0,(undefined (*) [32])"
     [+] Enter Serial: ",0x12);
8  local_d0 = 0;
9  local_c8 = 0xf;
10 local_e0[0] = (undefined8 ****)0x0;
11 FUN_140006640((undefined (*) [32])local_e0,(undefined (*) [32])"
     [!] Invalid Serial\n",0x13);
12 local_f0 = 0;
13 local_e8 = 0xf;
14 local_100[0] = (undefined8 ****)0x0;
15 FUN_140006640((undefined (*) [32])local_100,(undefined (*) [32])"
     [!] Correct Serial\n",0x13);
16 local_90 = 0;
17 local_88 = 0xf;
18 local_a0[0] = (undefined (*) [32])0x0;
```

Listing 2: ghidra output

It's not very clear why is here several function calls (line 7, 11, 15) that perform basic terminal output routine, but okay, let's get going. Stepping along the assembly I stumble upon interrupt in kernel32.dll that activates terminal input routine:

```
1 mov rcx,qword ptr ss:[rsp+38]
2 lea r9,qword ptr ss:[rsp+B8]
3 and qword ptr ss:[rsp+20],0
4 mov r8d,ebp
5 mov rdx,r15
6 call qword ptr ds:[<&ReadFile>]
7 test eax,eax
8 je crackme.7FF77AA79885
```

Listing 3: interrupt

At this point I noticed that generated serial is already lies somewhere at the bottom of stack:

```
00000012EA87FB68  00000012EA961EC0  "2562CFAD35C3B78DE3B92D913E"
```

Notice that the length of our key is 13 hexpairs which has same length as "KeyfrAQBc8Wsa". After entering serial key that I got from stack, the program sanitizes input:

- substitute \r with \n

- cut non-printable characters

```
mov dword ptr ss:[rsp+30],r8d
mov qword ptr ss:[rsp+28],rdx           ; [rsp+28]:"2562
    CFAD35C3B78DE3B92D913E\r\n"
lea rax,qword ptr ss:[rsp+50]
...
mov r8d,ebp
mov rdx,r15                             ; r15:"2562
    CFAD35C3B78DE3B92D913E\n\n"
call qword ptr ds:[<&ReadFile>]
...
mov rax,qword ptr ds:[rdi]              ;rax:"2562
    CFAD35C3B78DE3B92D913E", rdi:"2562CFAD35C3B78DE3B92D913E"
mov byte ptr ds:[rax+rcx],r8b           ; rcx is shift from string
    beginning here
mov byte ptr ds:[rax+rcx+1],0
```

Listing 4: Sanitisation routine

```
cmovae rdx,qword ptr ss:[rbp-58]        ; here lies generated
    license key
lea rcx,qword ptr ss:[rbp-78]           ; string x that equal to
    string  i entered
mov rbx,qword ptr ss:[rbp-78]           ; [rbp-78]:"2562
    CFAD35C3B78DE3B92D913E"
mov rdi,qword ptr ss:[rbp-60]
cmp rdi,10
cmovae rcx,rbx                          ; rcx:"2562
    CFAD35C3B78DE3B92D913E", rbx:"2562CFAD35C3B78DE3B92D913E"
mov r8,qword ptr ss:[rbp-68]            ; the value here is
    provided key length
cmp r8,qword ptr ss:[rbp-48]            ; the value here is
    generated key length
jne crackme.7FF77AA34C1E
call crackme.7FF77AA5AD30               ; strcmp, first buffer is
    user provided, second generated key
```

Listing 5: Check that key fits by length

In general - there is nothing exceptional. This part of code takes both string objects first checks if they have same length, if so - strcmp them. It is curious that in the strcmp function there is additional length check against 8 value and then jump a bit further. I have no idea for what kind of stuff this is done, would be interesting to learn:

```
sub rdx,rcx                             ; two cstring buffers
cmp r8,8                                ; number of chars to
    compare (it is much less that previous value...)
jb crackme.7FF77AA5AD5B
```

Listing 6: strcmp()

if returned value of strcmp function is not zero, then send user to invalid serial message:

```
test eax,eax                            ; if strcmp returned not
    zero, go away
jne crackme.7FF77AA34C1E
mov rax,qword ptr ds:[<&FatalExit>]
```

Listing 7: jump to fail/win message

Okay, now that we know there is nothing to catch here, lets return to the process of generating serial key - remember our function call at line 19 in the first listing? let's study it, guess I'll find something useful there.

2 Serial key generation

2.1 backtracing

Backtracing with x64dbg from previous code section gradually led me to the part of code where the serial key appeared first of all

```
mov rax,qword ptr ds:[rdi]
mov rcx,rdi
call qword ptr ds:[rax+150]             ; second one
xor r9d,r9d
mov byte ptr ss:[rsp+20],1
mov r8,rbx
mov rdx,r15
mov rcx,rax
mov r10,qword ptr ds:[rax]
call qword ptr ds:[r10+38]              ; candidate on role of
    generator of serial key
add r12,rbx
sub rbp,rbx
jne crackme.7FF77AA3E0B0
mov r15,qword ptr ss:[rsp+30]
mov r14,qword ptr ss:[rsp+38]           ; write pointer to
    serialkey here
```

Listing 8: There are two candidates on 'generator' role

Further backtracing of these two functions led me to spot where the serial key is being generated:

```
mov rsi,rdx                             ; move key
lea r9,qword ptr ds:[rcx+rdi*8]         ; r9:"KeyfrAQBc8Wsa", shift
...
mov rax,qword ptr ds:[r9+rsi]           ; rax contains 'KeyfrAQB'
xor qword ptr ds:[r9],rax               ;
...
mov eax,dword ptr ds:[rsi+r9]           ; eax contains 'c8Ws'
xor dword ptr ds:[r9],eax
...
movzx ecx,byte ptr ds:[rdx+rax]         ; ecx contains 'a'
xor byte ptr ds:[rax],cl                ; xor last byte
```

Listing 9: xoring first part of serial 'base' with key

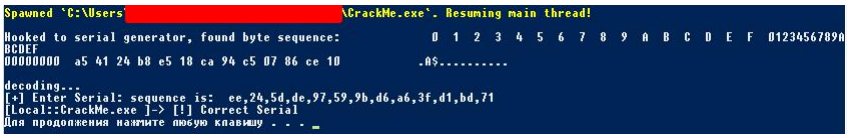

Figure 1: solved challenge

2.2 Hooking with frida

Great! Now let's extract this key from memory, consider the following code:

```
var mainModule = Process.enumerateModules()[0] // get main module info

/**
 * 4C 8D 0C F9 48 2B F1 4C 8B DB 4C 2B DF 48 8B FB 0F 1F 84 00 00 00 00 00 - byte sequence where i hook, in case you will want to take a look
 */
var hookAddress = mainModule.base.add(0x1E67C) // address where we can catch unxored serial base
var hooked = 0

// console.log("CrackMe.exe addres and hook address:\n")
// console.log(mainModule.base)
// console.log(hookAddress)
Interceptor.attach(hookAddress, {
    onEnter(args) {
        if (hooked == 0) {
            var pToDecode = new NativePointer(this.context.r9)
            var toDecode = pToDecode.readByteArray(13)
            console.log("\nHooked to serial generator, found byte sequence: ", toDecode)
            console.log("\ndecoding...")
            var serialEncoded = toDecode?.unwrap().readByteArray(13)
            var serialDecoded = []
            const xorKey = [0x4B, 0x65, 0x79, 0x66, 0x72, 0x41, 0x51, 0x42, 0x63, 0x38, 0x57, 0x73, 0x61]
            for (let index = 0; index < serialEncoded?.byteLength; index++) {
                serialDecoded.push((xorKey[index] ^ serialEncoded?.unwrap().add(index).readU8()).toString(16))

            }
        }
        console.log("sequence is: ", serialDecoded?.toString())
        hooked += 1
    }
});
```

Listing 10: Hooking to function with frida script

3 As a conclusion

Personally for me, there is one unsolved bit of puzzle - where does the serial base come from? Probably it is generated based on thread id or milliseconds value, idk. This crackme was quite challenging, it required several days of debugging to complete. Let me know in GitHub issues section or mail if you know how the base value is generated - the mail is available on my github page.
Oh, yeah, there is another research from other reverse engineer, take a look if you are interested in further research.
Thanks for your attention.