1 Preface

Greetings, in this article I will reverse engineer this crackme. The rules are to find how the serial key is being generated (Honestly, I'm not sure because the rules weren't strictly defined on the author's page), let's get started.

1.1 Toolset

- x64dbg
- ghidra/rizin/binja/ida
- Frida

I'll be using x64dbg to peek a look into the insides of the runtime program. It has proven itself worthy in the past, and it's actively developed by its maintainer, so as usual, x64dbg is my choice. I've already been using ghidra for some time in the past, but I prefer TUI tools in my researches, as well as reading pure assembler listing over decompiled code. Nonetheless, today I decided to give ghidra decompiler a try. Frida is a world-popular dynamic instrumentation tool, and it is quite easy to set it up and running, so for runtime hooking, I'll be using this tool.

1.2 First look

Let's run our binary file to take a look at it and get a general understanding of what we are dealing with:

Figure 1: first run

Okay, so we need to enter some kind of security key to pass the challenge. Let's dive into assembly code.

2 Finding clues

2.1 Input

At this stage, I tried to locate what comparison rules are and is there a way to fool them. The following assembler listings are somehow related to this process. I will on comment the most significant parts.

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

Listing 1: Begining of main function

Of interesting: note that from this point, we can see from which location our serial key is coming from. We will return to the 19th line later after we investigate input handling. Let's take a look at the main function using 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);
```

Listing 2: ghidra output

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

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

Listing 3: interrupt

At this point, I noticed that the generated serial has been already lying somewhere at the bottom of the stack:

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

Note that the length of our key is 13 hex pairs which has the same length as "KeyfrAQBc8Wsa". After entering the serial key that I obtained from the stack, the program sanitizes the 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

2.2 comparison rules

Further steps led me to this part of the main function:

```
cmovae rdx,qword ptr ss:[rbp-58]         ; here lies generated
    license key
lea rcx,qword ptr ss:[rbp-78]            ; string that 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               ; memcmp, 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 the code takes both string objects first and checks if they have the same length, if so - memcmp them. It is curious that in the memcmp function there is an additional length check against the "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: memcmp()

If the returned value of the memcmp function is not zero, then send the user to invalid serial message:

```
test eax,eax                            ; if memcmp 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, let's return to the process of generating the serial key - remember our function call at line 19 in the first listing? Let's study it, guess I'll find something useful there.

3 Serial key generation

3.1 backtracing

Backtracing using a debugger from the previous section gradually led me to the part where the serial key appeared first:

```
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 tracing 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

Summarizing gathered information from the listing:

0xc4	0xa9	0xf9	0xe9	0xb3	0xe2	0xe7	0xf8	0x38	0xa8	0xff	0x25	0xd9

example of random serial 'base' value

XOR

0x4b	0x65	0x79	0x66	0x72	0x41	0x51	0x42	0x63	0x38	0x57	0x73	0x61

"KeyfrAQBc8Wsa"

IS

0x8f	0xcc	0x80	0x8f	0xc1	0xa3	0xb6	0xba	0x5b	0x90	0xa8	0x56	0xb8

resulting serial key

Knowing that the algorithm performs a simple XOR operation, we can easily generate our key during runtime. The only thing we need to know is a random key which is pretty easy to obtain using Frida.

3.2 Hooking

Extracting this key from memory is quite a simple task since all we need to know is the offset of the place where we want to hook to and the base address of CrackMe.exe in virtual 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

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, 0x0x51, 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

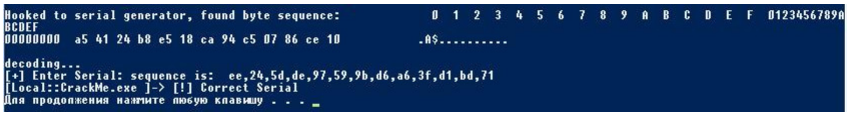

Figure 2: solved challenge

4 As a conclusion

The main challenge in this binary was that there is plenty of code that is not directly related to the information of our interest, so it requires quite a lot of time to find the trails of the code that leads us to a serial key generator routine. There was a moment where I almost caught the genuine sequence generator, but it appeared to be a simple hex-to-ascii translator. At that moment, I've already spent quite a lot of time writing Frida hook, so be cautious not to waste resources on meaningless things.
Personally, for me, there is one unsolved bit of the puzzle - where does the serial base key come from? It is probably generated based on thread ID or millisecond value.
This crackme was quite challenging because it took me several days of debugging to complete it. Let me know in GitHub issues section or mail if you understand how the base value is generated - the mail is available at my GitHub page.
Oh, yeah, there is another study from another reverse engineer, take a look if you are interested in further research.
Thanks for your attention.