FentCat - Assembler Crackme — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/68fce1922d267f28f69b783a
Author: *FentCat*
Write-up by: *SenorGPT*
Tools used: *CFF Explorer*, *x32dbg*

Platform	Difficulty	Quality	Arch	Language
Windows	2.0	3.5	x86	Assembler

Status: Complete
Goal: Document a clean path from initial recon → locating key-check logic → validation/reversal strategy

FentCat - Assembler Crackme — Reverse Engineering Write-up

- 1. Executive Summary
- 2. Target Overview
  - 2.1 UI / Behaviour
  - 2.2 Screens
    - Start-up
    - Failure case
- 3. Tooling & Environment
- 4. Static Recon
  - 4.1 File & Headers
  - 4.2 Imports / Exports
    - 4.2.1 KERNEL32.dll
- 5. Dynamic Analysis
  - 5.1 Baseline Run
  - 5.2 String Driven-Entry
- 6. Validation Path
- 7. Conclusion

1. Executive Summary

This crackme is a straightforward *x86* Windows console binary written in assembler that asks the user for a password, performs a series of length and content checks, and then either prints "*Welcome :O*" or "*Authentication Failed*". There are no real packing or heavy anti-debug tricks in play; the "*Warning: System integrity check running...*" message is purely cosmetic, and common debugger APIs such as `IsDebuggerPresent` are not imported.

My approach was to use string references as an entry point into the code, identify the main input/validation logic around `ReadConsoleA`, and then trace the control flow into the comparison function at `0x7310B8`. From there, I analyzed how the program uses global buffers for the input, the length field, and a hardcoded byte table at `crackme.7321DB` to compare the first eight bytes of the user input against a static reference sequence.

The key outcome is that the password check ultimately boils down to an 8-byte memcmp-style loop against the global data at `crackme.7321DB`. The length gate in `main` requires a *7-* or *8-*character input (accounting for the `\r\n` added by `ReadConsoleA`), but the comparison routine always iterates exactly *8 bytes* and only accepts the specific sequence `@CBEDGFI`. This provides a clean and deterministic way to recover the correct password without brute force.

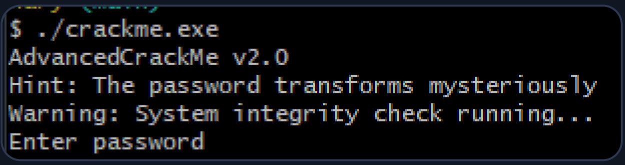
2. Target Overview

2.1 UI / Behaviour

- Inputs: **Accepts user input for a password.**
- Outputs: "*AdvancedCrackMe v2.0*", "*Hint: The password transforms mysteriously*", "*Warning: System integrity check running...*", "*Enter password*", "*Authentication Failed*"
- Expected protection level: Assume that there is some kind of anti-debugging due to the "*Warning: System integrity check running...*" message.

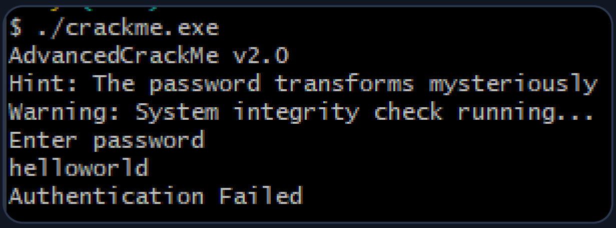
2.2 Screens

Start-up



```
$ ./crackme.exe
AdvancedCrackMe v2.0
Hint: The password transforms mysteriously
Warning: System integrity check running...
Enter password
```

Failure case



```
$ ./crackme.exe
AdvancedCrackMe v2.0
Hint: The password transforms mysteriously
Warning: System integrity check running...
Enter password
helloworld
Authentication Failed
```

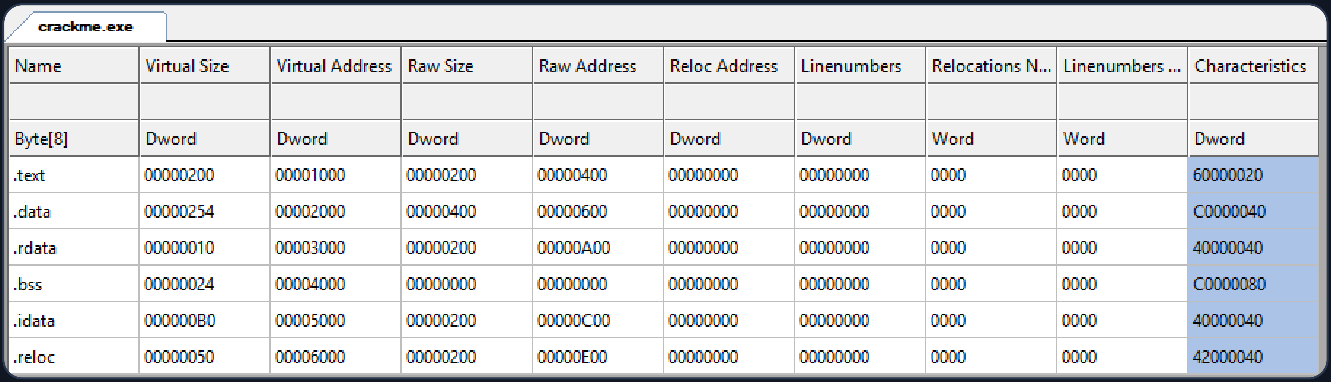
3. Tooling & Environment

- OS: *Windows 11*
- Debugger: *x32dbg*
- Decompiler (if applicable):

- Static tools: *CFF Explorer*

4. Static Recon

4.1 File & Headers



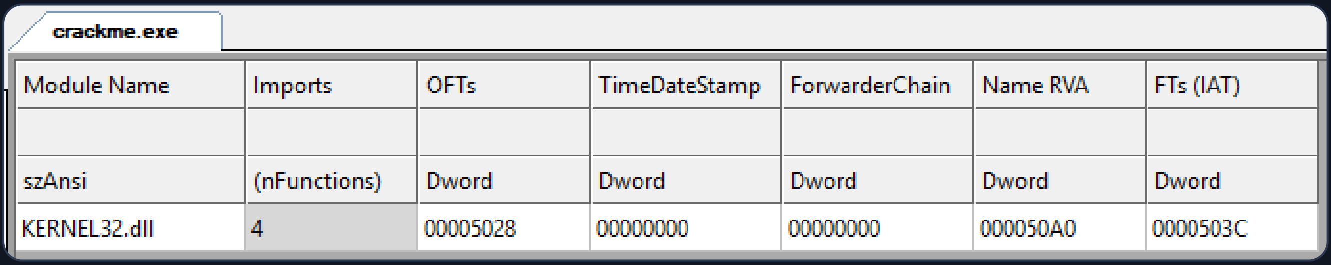
crackme.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00000200	00001000	00000200	00000400	00000000	00000000	0000	0000	60000020
.data	00000254	00002000	00000400	00000600	00000000	00000000	0000	0000	C0000040
.rdata	00000010	00003000	00000200	00000A00	00000000	00000000	0000	0000	40000040
.bss	00000024	00004000	00000000	00000000	00000000	00000000	0000	0000	C0000080
.idata	000000B0	00005000	00000200	00000C00	00000000	00000000	0000	0000	40000040
.reloc	00000050	00006000	00000200	00000E00	00000000	00000000	0000	0000	42000040

Notes:

- Architecture: *32-bit x86*, Windows console subsystem.
- Compiler hints: Small import table and straightforward control flow strongly suggest hand-written assembly or MASM/TASM-style code rather than a high-level language compiler.
- Packing/obfuscation signs: No section anomalies, no suspicious high-entropy sections, and imports are visible and usable. There are no signs of common packers or obfuscators.

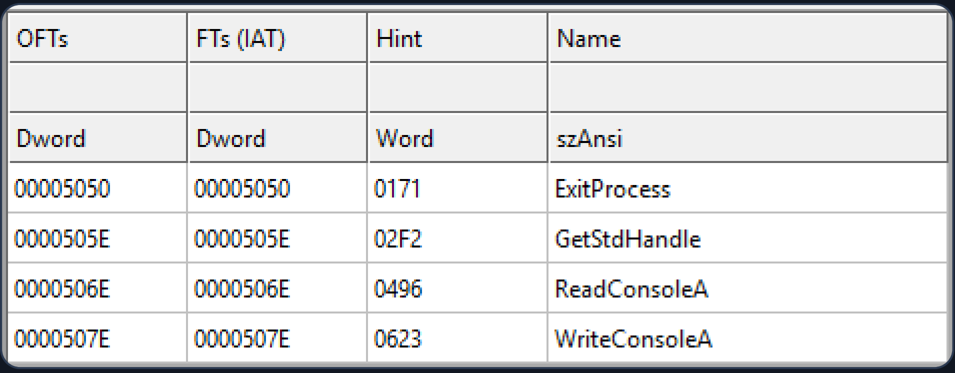
4.2 Imports / Exports



crackme.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	4	00005028	00000000	00000000	000050A0	0000503C

4.2.1 KERNEL32.dll



OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
00005050	00005050	0171	ExitProcess
0000505E	0000505E	02F2	GetStdHandle
0000506E	0000506E	0496	ReadConsoleA
0000507E	0000507E	0623	WriteConsoleA

Nothing stands out as any immediate anti-debugging calls.

---

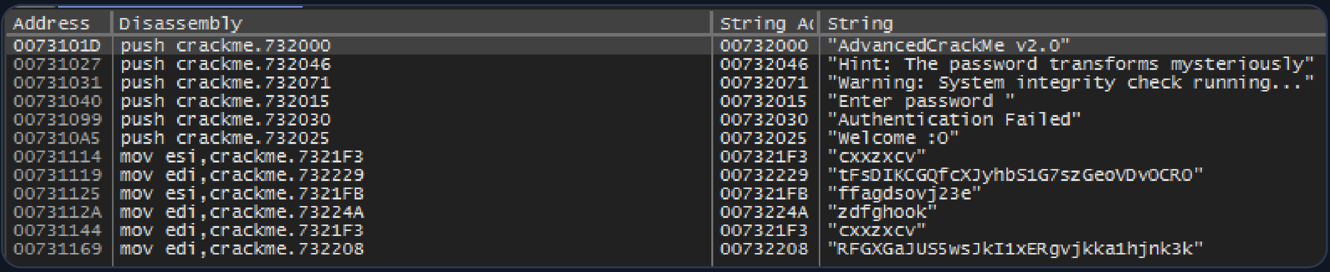
5. Dynamic Analysis

5.1 Baseline Run

Starting the program in *x32dbg* yields no immediate or obvious signs of any anti-debugging logic.

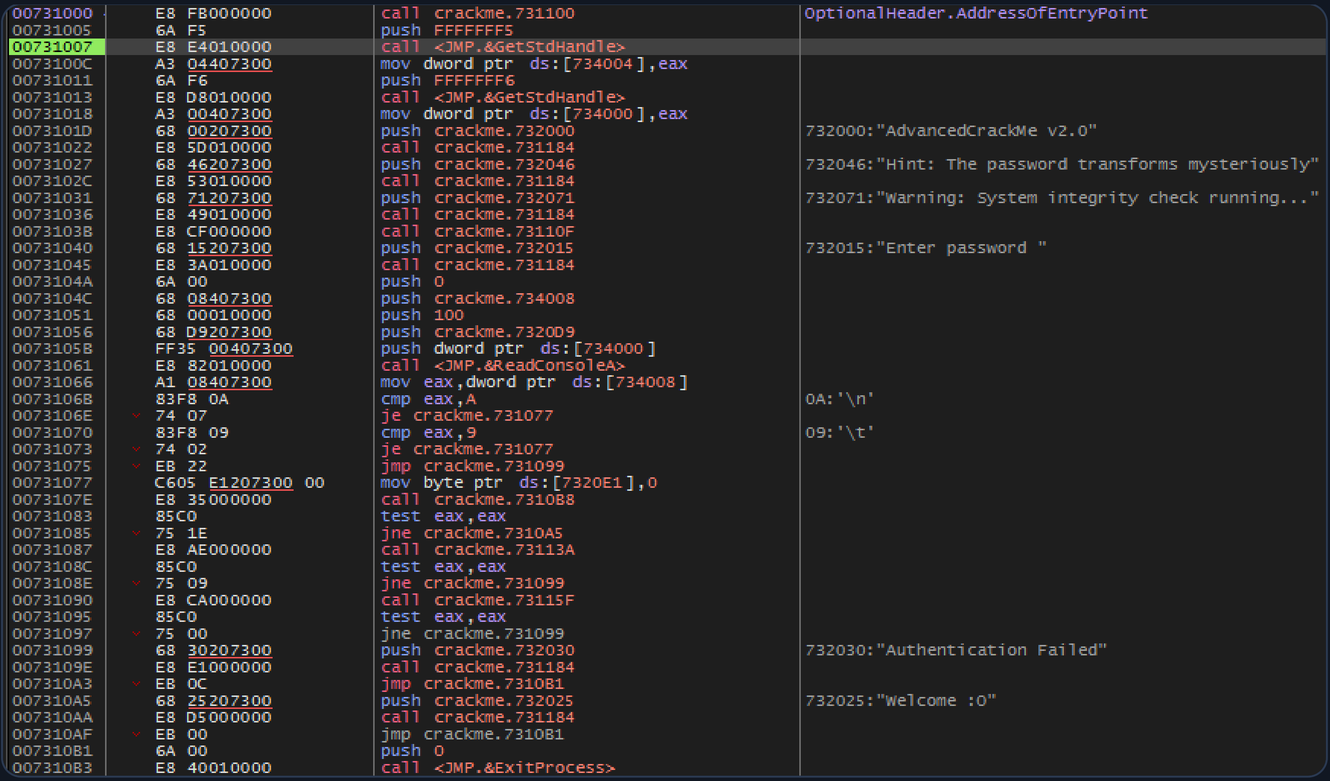
5.2 String Driven-Entry

Searching for string references within the target *Portable Executable* (*PE*) yields the following results.



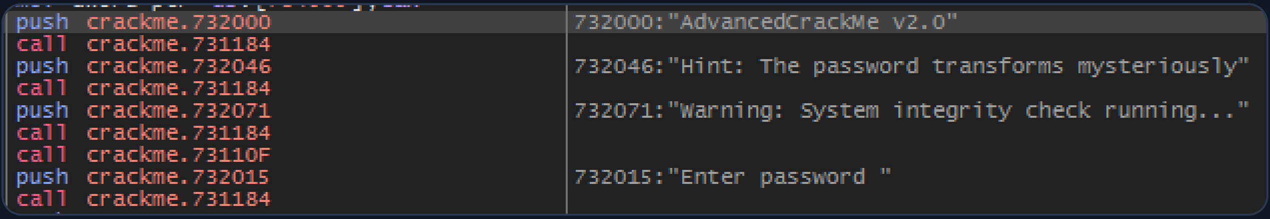
```
Address  Disassembly                 String Ad String
0073101D push crackme.732000         00732000  "AdvancedCrackMe v2.0"
00731027 push crackme.732046         00732046  "Hint: The password transforms mysteriously"
00731031 push crackme.732071         00732071  "Warning: System integrity check running..."
00731040 push crackme.732015         00732015  "Enter password "
00731099 push crackme.732030         00732030  "Authentication Failed"
007310A5 push crackme.732025         00732025  "Welcome :O"
00731114 mov esi,crackme.7321F3      007321F3  "cxxzxcv"
00731119 mov edi,crackme.732229      00732229  "tFsDIKCGQfcXJyhbS1G7szGeoVDvOCRO"
00731125 mov esi,crackme.7321FB      007321FB  "ffagdsovj23e"
0073112A mov edi,crackme.73224A      0073224A  "zdfghook"
00731144 mov edi,crackme.7321F3      007321F3  "cxxzxcv"
00731169 mov edi,crackme.732208      00732208  "RFGXGaJUS5wsJkI1xERgvjkka1hjnk3k"
```

Double clicking on the string reference for "*Enter password* " brings me into the disassembly view where I start to poke and prod around. I land on what seems to be the `main` function. The methods `GetStdHandle` and `ReadConsoleA` from `KERNEL32.dll` are observable here.



```
00731000   E8 FB000000        call crackme.731100               OptionalHeader.AddressOfEntryPoint
00731005   6A F5              push FFFFFFF5
00731007   E8 E4010000        call <JMP.&GetStdHandle>
0073100C   A3 04407300        mov dword ptr  ds:[734004],eax
00731011   6A F6              push FFFFFFF6
00731013   E8 D8010000        call <JMP.&GetStdHandle>
00731018   A3 00407300        mov dword ptr  ds:[734000],eax
0073101D   68 00207300        push crackme.732000               732000:"AdvancedCrackMe v2.0"
00731022   E8 5D010000        call crackme.731184
00731027   68 46207300        push crackme.732046               732046:"Hint: The password transforms mysteriously"
0073102C   E8 53010000        call crackme.731184
00731031   68 71207300        push crackme.732071               732071:"Warning: System integrity check running..."
00731036   E8 49010000        call crackme.731184
0073103B   E8 CF000000        call crackme.73110F
00731040   68 15207300        push crackme.732015               732015:"Enter password "
00731045   E8 3A010000        call crackme.731184
0073104A   6A 00              push 0
0073104C   68 08407300        push crackme.734008
00731051   68 00010000        push 100
00731056   68 D9207300        push crackme.7320D9
0073105B   FF35 00407300      push dword ptr  ds:[734000]
00731061   E8 82010000        call <JMP.&ReadConsoleA>
00731066   A1 08407300        mov eax,dword ptr  ds:[734008]
0073106B   83F8 0A            cmp eax,A                         0A:'\n'
0073106E   74 07              je crackme.731077
00731070   83F8 09            cmp eax,9                         09:'\t'
00731073   74 02              je crackme.731077
00731075   EB 22              jmp crackme.731099
00731077   C605 E1207300 00   mov byte ptr  ds:[7320E1],0
0073107E   E8 35000000        call crackme.7310B8
00731083   85C0               test eax,eax
00731085   75 1E              jne crackme.7310A5
00731087   E8 AE000000        call crackme.73113A
0073108C   85C0               test eax,eax
0073108E   75 09              jne crackme.731099
00731090   E8 CA000000        call crackme.73115F
00731095   85C0               test eax,eax
00731097   75 00              jne crackme.731099
00731099   68 30207300        push crackme.732030               732030:"Authentication Failed"
0073109E   E8 E1000000        call crackme.731184
007310A3   EB 0C              jmp crackme.7310B1
007310A5   68 25207300        push crackme.732025               732025:"Welcome :O"
007310AA   E8 D5000000        call crackme.731184
007310AF   EB 00              jmp crackme.7310B1
007310B1   6A 00              push 0
007310B3   E8 40010000        call <JMP.&ExitProcess>
```

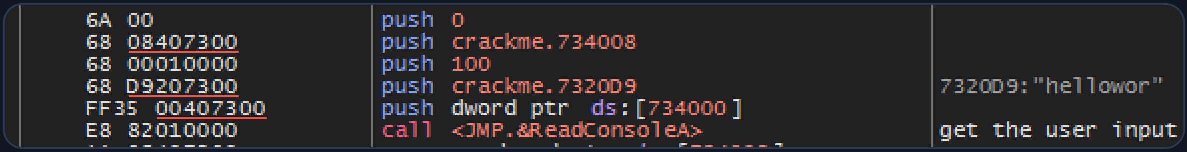
These calls seem to be responsible for outputting the string references onto the console.



```
push  crackme.732000      732000:"AdvancedCrackMe v2.0"
call  crackme.731184
push  crackme.732046      732046:"Hint: The password transforms mysteriously"
call  crackme.731184
push  crackme.732071      732071:"Warning: System integrity check running..."
call  crackme.731184
call  crackme.73110F
push  crackme.732015      732015:"Enter password "
call  crackme.731184
```

The `push` instructions are loading the string reference as a parameter for their following `call` instruction, which I presume is a `printf` style call of sorts.

The next few `push` instructions are preparing the parameters for the `ReadConsoleA` call.



```
6A 00            push  0
68 08407300      push  crackme.734008
68 00010000      push  100
68 D9207300      push  crackme.7320D9              7320D9:"hellowor"
FF35 00407300    push  dword ptr  ds:[734000]
E8 82010000      call  <JMP.&ReadConsoleA>         get the user input
```

```
BOOL ReadConsoleA(
    HANDLE  hConsoleInput,
    LPVOID  lpBuffer,
    DWORD   nNumberOfCharsToRead,
    LPDWORD lpNumberOfCharsRead,
    LPVOID  pInputControl
);
```

So, in the assembly code, the parameters can be labelled as the following.

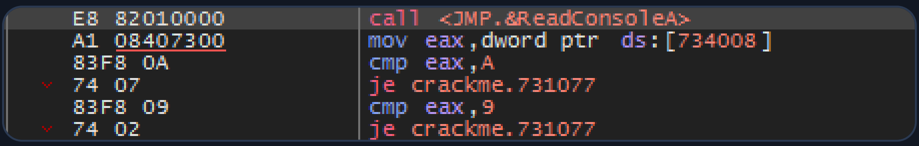
```
push 0                          ; lpReserved
push crackme.734008             ; LPDWORD lpNumberOfCharsRead
push 100                        ; nNumberOfCharsToRead
push crackme.7320D9             ; lpBuffer
push dword ptr ds:[734000]      ; hConsoleInput
```

That means that the user input is being stored in `crackme.7320D9` and the length of the user input in `crackme.734008`.

---

6. Validation Path

Right after the call to `ReadConsoleA`, the input length is loaded into `EAX` and then compared against `0xA`. At first I thought this call was checking for empty input. After further analysis, I figured out it was actually comparing the user input length from `ReadConsoleA` against `0xA`.



```
  E8 82010000   call <JMP.&ReadConsoleA>
  A1 08407300   mov eax,dword ptr ds:[734008]
  83F8 0A       cmp eax,A
  74 07         je crackme.731077
  83F8 09       cmp eax,9
  74 02         je crackme.731077
```

At first glance, it might seem that it is comparing the user input length against `0xA` (10) but it is important to keep in mind that in line mode, `ReadConsoleA` includes the `CR+LF` from you pressing Enter.

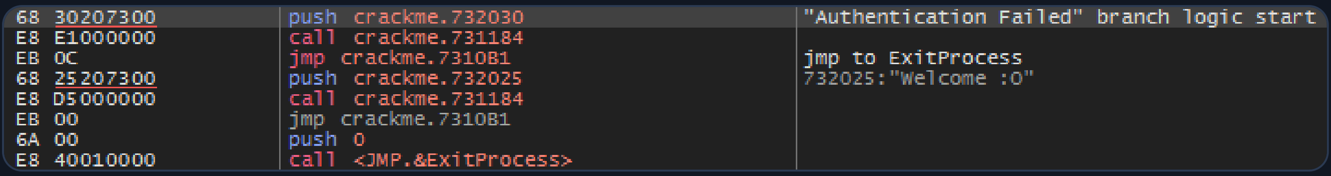
When you type in the console:

```
abc123⏎
```

what actually gets put into the buffer is (in hex):

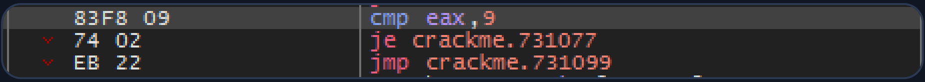
```
61 62 63 31 32 33 0D 0A
 a  b  c  1  2  3 \r \n
```

So, the `cmp EAX, 0xA` instruction is checking if the user input is *8* characters long, not *10*. If the check fails, the logic jumps to `crackme.731099` which is the "*Authentication Failed*" logic.



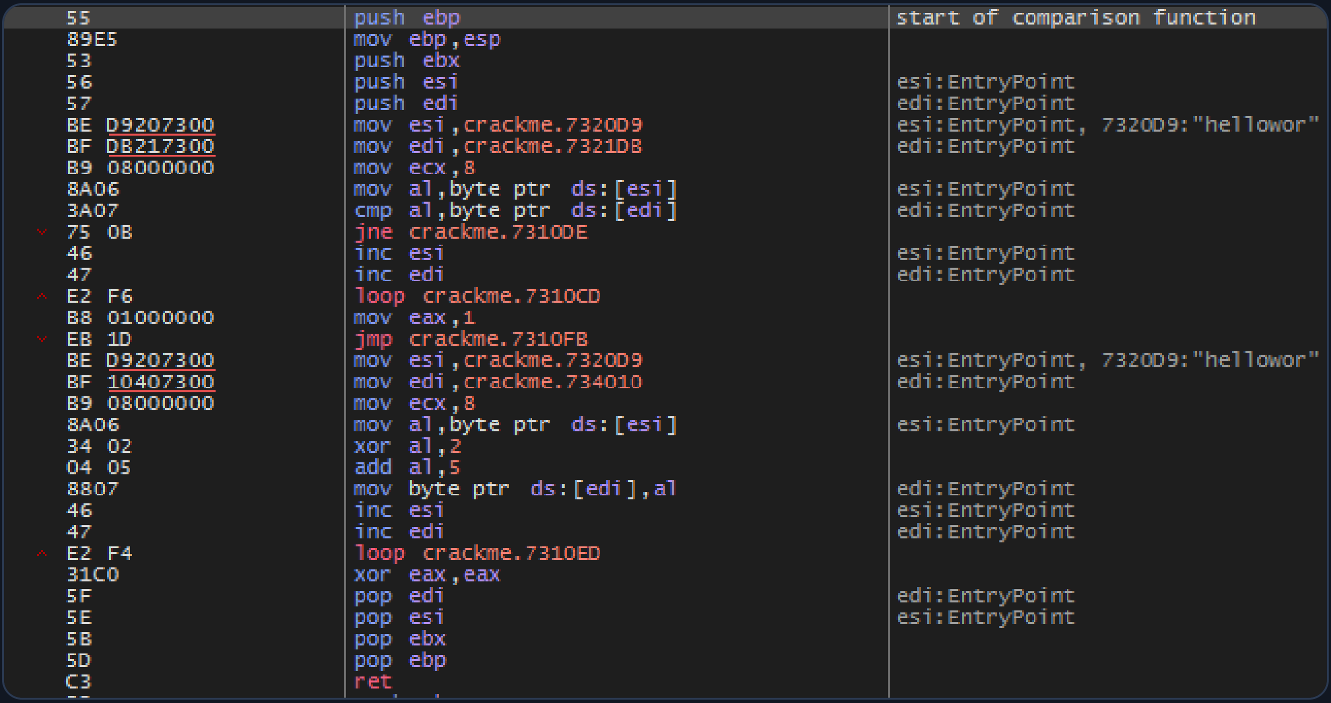
```
68 30207300     push  crackme.732030          "Authentication Failed" branch logic start
E8 E1000000     call  crackme.731184
EB 0C           jmp  crackme.7310B1           jmp to ExitProcess
68 25207300     push  crackme.732025          732025:"Welcome :O"
E8 D5000000     call  crackme.731184
EB 00           jmp  crackme.7310B1
6A 00           push  0
E8 40010000     call  <JMP.&ExitProcess>
```

Following the `cmp EAX, 0xA` instruction is another `cmp` instruction, instead this time comparing `EAX` to `0x9` (9) - `cmp EAX, 0x9`.



```
83F8 09     cmp  eax,9
74 02       je  crackme.731077
EB 22       jmp  crackme.731099
```

So these two `cmp` instructions are checking if the user input is *7* or *8* characters long, if it's not it jumps to the aforementioned "*Authentication Failed*" logic. If the length conditions are met, the logic jumps to further input validation which seems to be a call to `crackme.7310B8`.
Adding a breakpoint on that call and stepping into it reveals the following.

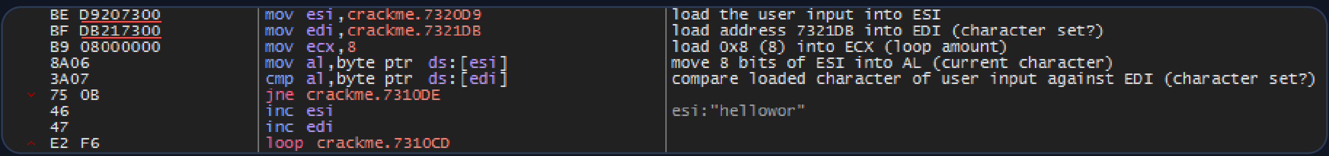


```
55              push  ebp                       start of comparison function
89E5            mov  ebp,esp
53              push  ebx
56              push  esi                       esi:EntryPoint
57              push  edi                       edi:EntryPoint
BE D9207300     mov  esi,crackme.7320D9         esi:EntryPoint, 7320D9:"hellowor"
BF DB217300     mov  edi,crackme.7321DB         edi:EntryPoint
B9 08000000     mov  ecx,8
8A06            mov  al,byte ptr  ds:[esi]      esi:EntryPoint
3A07            cmp  al,byte ptr  ds:[edi]      edi:EntryPoint
75 0B           jne  crackme.7310DE
46              inc  esi                        esi:EntryPoint
47              inc  edi                        edi:EntryPoint
E2 F6           loop  crackme.7310CD
B8 01000000     mov  eax,1
EB 1D           jmp  crackme.7310FB
BE D9207300     mov  esi,crackme.7320D9         esi:EntryPoint, 7320D9:"hellowor"
BF 10407300     mov  edi,crackme.734010         edi:EntryPoint
B9 08000000     mov  ecx,8
8A06            mov  al,byte ptr  ds:[esi]      esi:EntryPoint
34 02           xor  al,2
04 05           add  al,5
8807            mov  byte ptr  ds:[edi],al      edi:EntryPoint
46              inc  esi                        esi:EntryPoint
47              inc  edi                        edi:EntryPoint
E2 F4           loop  crackme.7310ED
31C0            xor  eax,eax
5F              pop  edi                        edi:EntryPoint
5E              pop  esi                        esi:EntryPoint
5B              pop  ebx
5D              pop  ebp
C3              ret
```

At the start of the function, it loads in the user input pointer into `ESI` and a global variable `crackme.7321DB` into `EDI`. Loading `crackme.7321DB` in the *Dump* reveals the following.

```
007321DB  40 43 42 45 44 47 46 49 32 31 33 35 34 36 38 37  @CBEDGFI21354687
007321EB  15 23 37 41 52 66 74 89 63 78 78 7A 78 63 76 00  .#7ARft.cxxzxcv.
007321FB  66 66 61 67 64 73 6F 76 6A 32 33 65 00 52 46 47  ffagdsovj23e.RFG
0073220B  58 47 61 4A 55 53 35 77 73 4A 6B 49 31 78 45 52  XGaJUS5wsJkI1xER
0073221B  67 76 6A 6B 6B 61 31 68 6A 6E 6B 33 6B 00 74 46  gvjkka1hjnk3k.tF
0073222B  73 44 49 4B 43 47 51 66 63 58 4A 79 68 62 53 31  sDIKCGQfcXJyhbS1
0073223B  47 37 73 7A 47 65 6F 56 44 76 4F 43 52 4F 00 7A  G7szGeoVDvOCRO.z
0073224B  64 66 67 68 6F 6F 6B 00 00 00 00 00 00 00 00 00  dfghook.........
```

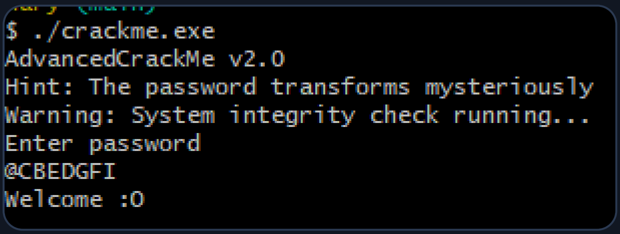
The function then proceeds to a loop which does the comparison checks. Checking one character at a time of the user input against the global characters at `crackme.7321DB`.



```
  BE D9207300   mov esi,crackme.7320D9         load the user input into ESI
  BF DB217300   mov edi,crackme.7321DB         load address 7321DB into EDI (character set?)
  B9 08000000   mov ecx,8                      load 0x8 (8) into ECX (loop amount)
  8A06          mov al,byte ptr ds:[esi]       move 8 bits of ESI into AL (current character)
  3A07          cmp al,byte ptr ds:[edi]       compare loaded character of user input against EDI (character set?)
˅ 75 0B         jne crackme.7310DE
  46            inc esi                        esi:"hellowor"
  47            inc edi
˄ E2 F6         loop crackme.7310CD
```

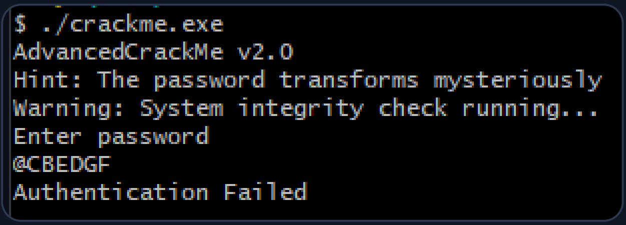
Each time the loop iterates, the `ESI` and `EDI` registers increment by 1, which is *1-byte* or *8-bits*. Which means that it is going straight through the character map and not jumping around.
Therefore, taking either the strings of `@CBEDGFI` or `@CBEDGF` should work.

Trying `@CBEDGFI`.



```
$ ./crackme.exe
AdvancedCrackMe v2.0
Hint: The password transforms mysteriously
Warning: System integrity check running...
Enter password
@CBEDGFI
Welcome :O
```

Trying `@CBEDGFI`.



```
$ ./crackme.exe
AdvancedCrackMe v2.0
Hint: The password transforms mysteriously
Warning: System integrity check running...
Enter password
@CBEDGF
Authentication Failed
```

Huh, looking back the comparison check within the `main` function it allowed either *7* or *8* characters to proceed into the validation function. But, the validation function always iterates and checks for 8 characters as indicated by the `mov ecx, 8` instruction.

7. Conclusion

This crackme turned out to be a clean example of classic *32-bit* Windows console reversing with hand-written assembly, rather than a heavily protected or obfuscated target. The program sets up console I/O via `GetStdHandle` and `ReadConsoleA`, enforces a simple length gate on the user's password (allowing only *7*- or *8*-character inputs when accounting for `\r\n`), and then delegates the actual validation to a small comparison function at `0x7310B8`.

Inside that function, the input buffer at `crackme.7320D9` is compared byte-by-byte against a global table at `crackme.7321DB` using a pointer-based loop. Because `ECX` is initialized to *8*, the function always performs eight comparisons and only returns success (`EAX = 1`) when all eight bytes match. Dumping the global table reveals that the first *eight bytes* are `40 43 42 45 44 47 46 49`, which correspond to the ASCII string `@CBEDGFI`. This is the only password that satisfies both the length gate and the comparison logic and results in the *"Welcome :O"* branch.

From a learning standpoint, this challenge was useful for reinforcing several core concepts:

- Understanding how *WinAPI* calls like `ReadConsoleA` use output parameters (`LPBuffer`, `LPDWORD lpNumberOfCharsRead`) and how that influences length checks.
- Recognizing global/static data (e.g., `crackme.7321DB`) versus stack-based locals or arguments.
- Reading pointer-based loops (`ESI`/`EDI` plus `inc` + `loop`) as a memcmp-style operation without an explicit index variable.
- Seeing how a seemingly flexible length check in `main` can still funnel into a strict fixed-length comparison deeper in the call graph.

The final solution is the recovered password:

```
@CBEDGFI
```