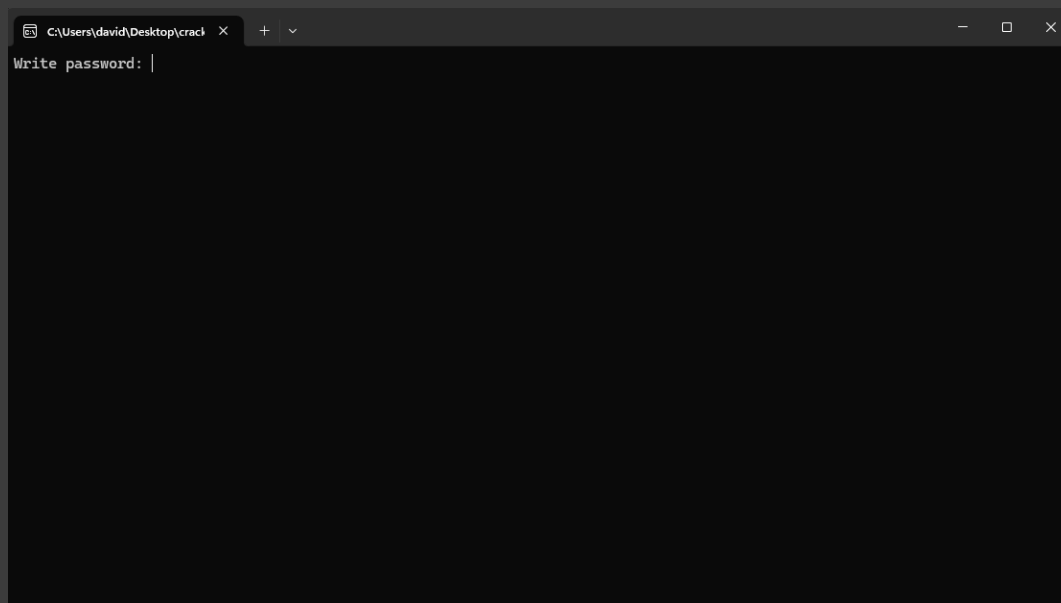


Crackme: <https://crackmes.one/crackme/69245c422d267f28f69b806e>

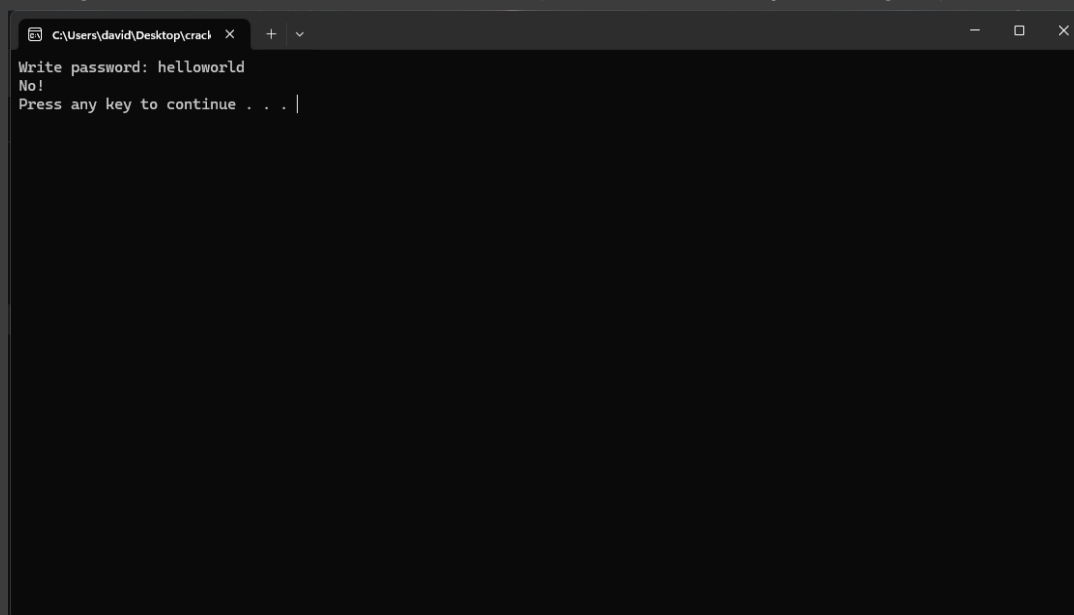
Author: vilxd Software Used: x64dbg Writeup By: SenorGPT

| | | | | |
|---------------------|--------------------|-----------------|--------------|--------------------|
| Platform Windows | Difficulty: 2.2 | Quality: 4.0 | Arch: x86 | Language: C/C++ |
|---------------------|--------------------|-----------------|--------------|--------------------|

Regular execution (startup):



Wrong answer provided - proceeded by termination on following key press:



Goal: recover the correct password string (no patching required to solve), using static and/or dynamic analysis.

The architecture on the crackme page states x86 although I needed to use x64dbg to analyze the file which points to the metadata being wrong on the page.

My first approach is to search for string references related to the CTF PE (Portable Executable) within x64dbg. To do this, I added a breakpoint before the entry of the executable.

| | | | | | | | | | | |
|------------------|--|-------------|-------------|------------|------------------|--------|---------|--------|------------|---------|
| Log | Notes | Breakpoints | Memory Map | Call Stack | SEH | Script | Symbols | Source | References | Threads |
| Address | Module/Label/Exception | State | Disassembly | Hits | Summary | | | | | |
| 00007F6680D71125 | <decodeme.exe.OptionalHeader.AddressOf | One-time | push rbp | 0 | entry breakpoint | | | | | |

Then going into Symbols and double clicking the CTF PE (decodeme.exe) to bring me into the CPU tab with the disassembly view. Since I am now in the target module, I can right click the disassembly empty space — Search For — Current Module — String References.

| CPU | Log | Notes | Breakpoints | Memory Map | Call Stack | SEH | Script | Symbols | Source | References | Threads | Handles |
|------------------------|-------------------------------------|------------------|---|------------|------------|-----|--------|---------|--------|------------|---------|---------|
| Strings (decodeme.exe) | | | | | | | | | | | | |
| Address | Disassembly | String Address | String | | | | | | | | | |
| 00007FF680D71634 | lea rdx,qword ptr ds:[7FF680D83000] | 00007FF680D83000 | "\\x02X" | | | | | | | | | |
| 00007FF680D71639 | lea rdx,qword ptr ds:[7FF680D83007] | 00007FF680D83007 | "%s" | | | | | | | | | |
| 00007FF680D7176C | lea rbp,qword ptr ds:[7FF680D83000] | 00007FF680D83000 | "\\x02X" | | | | | | | | | |
| 00007FF680D7199C | lea rcx,qword ptr ds:[7FF680D8300A] | 00007FF680D8300A | "Yes!" | | | | | | | | | |
| 00007FF680D719B0 | lea rcx,qword ptr ds:[7FF680D8300F] | 00007FF680D8300F | "No!" | | | | | | | | | |
| 00007FF680D71A0E | lea rcx,qword ptr ds:[7FF680D83019] | 00007FF680D83019 | "Write password: " | | | | | | | | | |
| 00007FF680D71A20 | lea rcx,qword ptr ds:[7FF680D83013] | 00007FF680D83013 | "pause" | | | | | | | | | |
| 00007FF680D71C9C | lea rax,qword ptr ds:[7FF680D83080] | 00007FF680D83080 | "Argument domain error (DOMAIN)" | | | | | | | | | |
| 00007FF680D71CA9 | lea rax,qword ptr ds:[7FF680D8309F] | 00007FF680D8309F | "Argument singularity (SIGERR)" | | | | | | | | | |
| 00007FF680D71CB6 | lea rax,qword ptr ds:[7FF680D830C0] | 00007FF680D830C0 | "Overflow range error (OVERFLOW)" | | | | | | | | | |
| 00007FF680D71CC3 | lea rax,qword ptr ds:[7FF680D830E0] | 00007FF680D830E0 | "Partial loss of significance (PLOSS)" | | | | | | | | | |
| 00007FF680D71CD0 | lea rax,qword ptr ds:[7FF680D83108] | 00007FF680D83108 | "Total loss of significance (TLOSS)" | | | | | | | | | |
| 00007FF680D71CEA | lea rax,qword ptr ds:[7FF680D83130] | 00007FF680D83130 | "The result is too small to be represented (UNDERFLOW)" | | | | | | | | | |
| 00007FF680D71D44 | lea rax,qword ptr ds:[7FF680D83178] | 00007FF680D83178 | "Unknown error" | | | | | | | | | |
| 00007FF680D71E87 | lea rax,qword ptr ds:[7FF680D831C0] | 00007FF680D831C0 | "matherr(): %s in %s(%g, %g) (retval=%g)\n" | | | | | | | | | |
| 00007FF680D71F84 | lea rax,qword ptr ds:[7FF680D831E0] | 00007FF680D831E0 | "Unknown pseudo runtime failure %d" | | | | | | | | | |
| 00007FF680D720A8 | lea rax,qword ptr ds:[7FF680D83200] | 00007FF680D83200 | "Address %p has no image-section" | | | | | | | | | |
| 00007FF680D722D0 | lea rax,qword ptr ds:[7FF680D83238] | 00007FF680D83238 | "VirtualQuery failed for %d bytes at address %p" | | | | | | | | | |
| 00007FF680D7240C | lea rax,qword ptr ds:[7FF680D83260] | 00007FF680D83260 | "VirtualProtect failed with code 0xxx" | | | | | | | | | |
| 00007FF680D72440 | lea rax,qword ptr ds:[7FF680D83298] | 00007FF680D83298 | "Unknown pseudo relocation protocol version %d.\n" | | | | | | | | | |
| 00007FF680D724C4 | lea rax,qword ptr ds:[7FF680D832C8] | 00007FF680D832C8 | "Unknown pseudo relocation bit size %d.\n" | | | | | | | | | |
| 00007FF680D72AC5 | lea r9,qword ptr ds:[7FF680D83350] | 00007FF680D83350 | "%d bit pseudo relocation at %p out of range, targeting %p, yielding the value %p.\n" | | | | | | | | | |
| 00007FF680D7605E | lea rax,qword ptr ds:[7FF680D83356] | 00007FF680D83356 | "(nil)" | | | | | | | | | |
| 00007FF680D761A0 | lea rax,qword ptr ds:[7FF680D8335A] | 00007FF680D8335A | "nan" | | | | | | | | | |
| 00007FF680D7630C | lea rax,qword ptr ds:[7FF680D8335E] | 00007FF680D8335E | "inf" | | | | | | | | | |
| 00007FF680D77E3A | lea rax,qword ptr ds:[7FF680D83610] | 00007FF680D83610 | "inity" | | | | | | | | | |
| 00007FF680D7801F | lea rax,qword ptr ds:[7FF680D83618] | 00007FF680D83618 | "(null)" | | | | | | | | | |
| 00007FF680D79A20 | lea rdx,qword ptr ds:[7FF680D83626] | 00007FF680D83626 | "L'(null)" | | | | | | | | | |
| 00007FF680D79A7B | lea rdx,qword ptr ds:[7FF680D8362A] | 00007FF680D8362A | "NaN" | | | | | | | | | |
| 00007FF680D79B2F | lea rdx,qword ptr ds:[7FF680D83626] | 00007FF680D83626 | "inf" | | | | | | | | | |
| 00007FF680D79B86 | lea rdx,qword ptr ds:[7FF680D8362A] | 00007FF680D8362A | "NaN" | | | | | | | | | |
| 00007FF680D7ABFD | lea rax,qword ptr ds:[7FF680D837A0] | 00007FF680D837A0 | "Infinity" | | | | | | | | | |
| 00007FF680D7AC2E | lea rax,qword ptr ds:[7FF680D837A9] | 00007FF680D837A9 | "NaN" | | | | | | | | | |
| 00007FF680D7E231 | lea rdx,qword ptr ds:[7FF680D8390C] | 00007FF680D8390C | "inf" | | | | | | | | | |
| 00007FF680D7E258 | lea rdx,qword ptr ds:[7FF680D8390F] | 00007FF680D8390F | "inity" | | | | | | | | | |
| 00007FF680D7E287 | lea rdx,qword ptr ds:[7FF680D839E5] | 00007FF680D839E5 | "an" | | | | | | | | | |
| 00007FF680D80AE1 | lea rax,qword ptr ds:[7FF680D83AE0] | 00007FF680D83AE0 | "0123456789" | | | | | | | | | |
| 00007FF680D80B00 | lea rax,qword ptr ds:[7FF680D83AE8] | 00007FF680D83AE8 | "abcde" | | | | | | | | | |
| 00007FF680D80B1F | lea rax,qword ptr ds:[7FF680D83AF2] | 00007FF680D83AF2 | "ABCDEDEF" | | | | | | | | | |
| 00007FF680D81D7F | lea rcx,qword ptr ds:[7FF680D8302A] | 00007FF680D8302A | "Pa100-322-1L@101" | | | | | | | | | |
| 00007FF680D81DAE | lea rcx,qword ptr ds:[7FF680D83007] | 00007FF680D83007 | "%s" | | | | | | | | | |
| 00007FF680D81D0C | lea rcx,qword ptr ds:[7FF680D83013] | 00007FF680D83013 | "pause" | | | | | | | | | |
| 00007FF680D8322D | and byte ptr ds:[7FF680D832A3],ah | 00007FF680D832A3 | "seudo relocation bit size %d.\n" | | | | | | | | | |

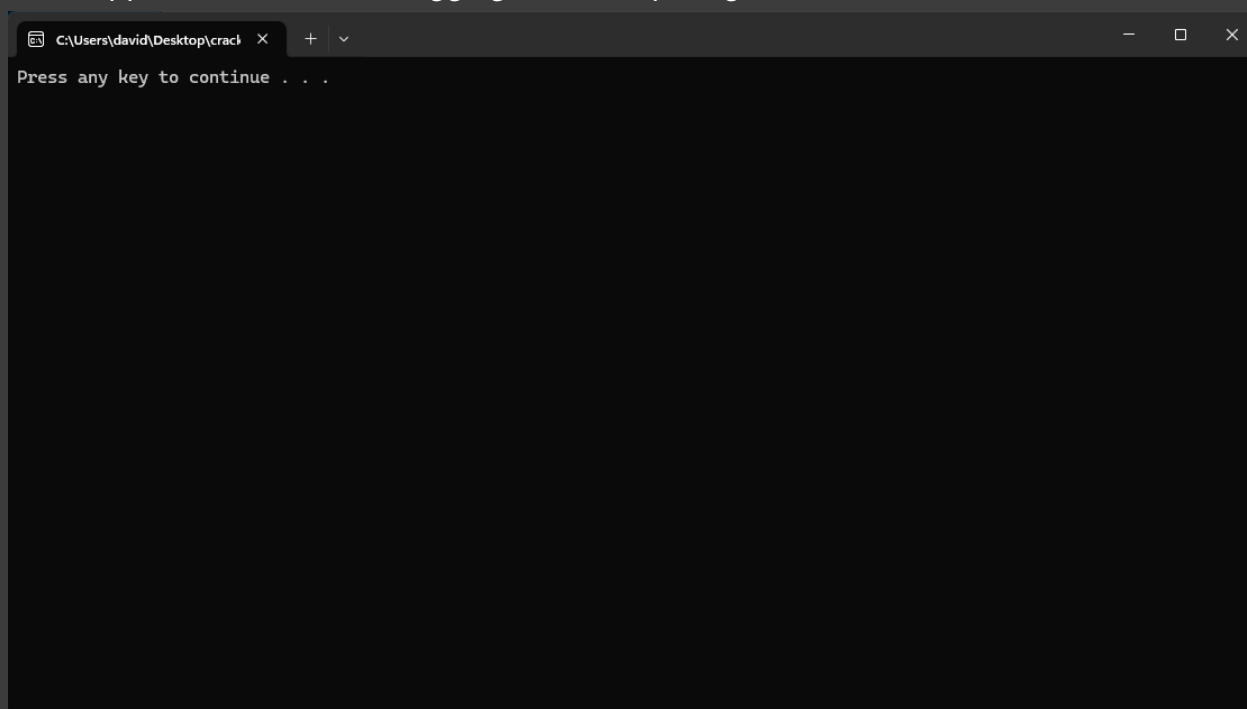
In this view I am able to see all the string references that the module defines and uses.

Currently, I am interested in the strings that I can reproduce - "Write password: ", "No!", and "Press any key to continue..." - as well as the "Yes!" string as I can infer that it might be the response the user receives if they enter in the correct flag/key.

It appears that "Press any key to continue..." belongs to cmd.exe, not the CTF PE module and this is why I am not able to see a string reference to it under the target module "decodeme.exe".

Here is where I made another discovery once I tried to just run the CTF under x64dbg.

There appears to be anti-debugging / anti-tampering code.



Time to add some more breakpoints! Since it appears that it is checking to see if a debugger is attached I will start by adding the following breakpoints:

1. bp kernel32.IsDebuggerPresent
2. bp kernel32.CheckRemoteDebuggerPresent
3. bp ntdll.NtQueryInformationProcess

```
Breakpoint at 00007FFFC16FD9A0 set!  
Breakpoint at 00007FFFC1705230 set!  
Breakpoint at 00007FFFC1F81F00 set!
```

Once added, I restart program execution. I got a few hits on ntdll.NtQueryInformationProcess, but nothing interesting.

Then, I got a break on kernel32.IsDebuggerPresent. Things are starting to get spicy.

From the call stack I am able to see that the target module is the one that is making the call to IsDebuggerPresent. Goodie, that is exactly what I want to see.

| Address | To | From | Size | Party | Comment |
|------------------|------------------|------------------|------|--------|---------------------------------|
| 000000E2C7FFF3D8 | 00007FF68DD71A0A | 00007FFFC16FD9A0 | 30 | User | kernel32.IsDebuggerPresent |
| 000000E2C7FFF408 | 00007FF68DD81DAB | 00007FF68DD71A0A | 380 | User | decodeme.00007FF68DD71A0A |
| 000000E2C7FFF788 | 00007FF68DD71340 | 00007FF68DD81DAB | 80 | User | decodeme.00007FF68DD81DAB |
| 000000E2C7FFF808 | 00007FF68DD71146 | 00007FF68DD71340 | 40 | User | decodeme.00007FF68DD71340 |
| 000000E2C7FFF848 | 00007FFFC16EE8D7 | 00007FF68DD71146 | 30 | System | decodeme.00007FF68DD71146 |
| 000000E2C7FFF878 | 00007FFFC1EAC53C | 00007FFFC16EE8D7 | 50 | System | kernel32.BaseThreadInitThunk+17 |
| 000000E2C7FFF8C8 | 0000000000000000 | 00007FFFC1EAC53C | | User | ntdll.RtlUserThreadStart+2C |

Double clicking on the frame underneath the kernel32.IsDebuggerPresent call brings me to the function that called it, which appears to be the main logic of the program.

| | | | |
|--------------------|------------------|--|-------------------------------------|
| 00007FF68D071A04 | FF15 56680100 | call qword ptr ds:[<IsDebuggerPresent>] | |
| Breakpoint Enabled | 85C0 | test eax, eax | |
| 00007FF68D071A0A | 75 12 | jne decode.7FF68D071A20 | |
| 00007FF68D071A0E | 48:8D0D 04160100 | lea rcx, qword ptr ds:[7FF68D083019] | 00007FF68D083019:"write password: " |
| 00007FF68D071A15 | 48:83C4 28 | add rsp, 28 | |
| 00007FF68D071A19 | E9 C2F8FFFF | jmp decode.7FF68D0715E0 | |
| 00007FF68D071A1E | 66:90 | nop | |
| 00007FF68D071A20 | 48:8D0D EC150100 | lea rcx, qword ptr ds:[7FF68D083013] | 00007FF68D083013:"pause" |
| 00007FF68D071A27 | 48:83C4 28 | add rsp, 28 | |
| 00007FF68D071A2B | E9 90020100 | jmp <JMP.&system> | |
| 00007FF68D071A30 | FF25 72680100 | jmp qword ptr ds:[<_C_specific_handler>] | JMP.&_C_specific_handler |
| 00007FF68D071A36 | 90 | nop | |
| 00007FF68D071A37 | 90 | nop | |
| 00007FF68D071A38 | FF25 42680100 | jmp qword ptr ds:[<Sleep>] | JMP.&Sleep |
| 00007FF68D071A3E | 90 | nop | |
| 00007FF68D071A3F | 90 | nop | |
| 00007FF68D071A40 | FF25 32680100 | jmp qword ptr ds:[<SetUnhandledExceptionFilter>] | JMP.&SetUnhandledExceptionFilter |
| 00007FF68D071A46 | 90 | nop | |
| 00007FF68D071A47 | 90 | nop | |
| 00007FF68D071A48 | FF25 12680100 | jmp qword ptr ds:[<IsDebuggerPresent>] | JMP.&IsDebuggerPresent |
| 00007FF68D071A4E | 90 | nop | |
| 00007FF68D071A4F | 90 | nop | |

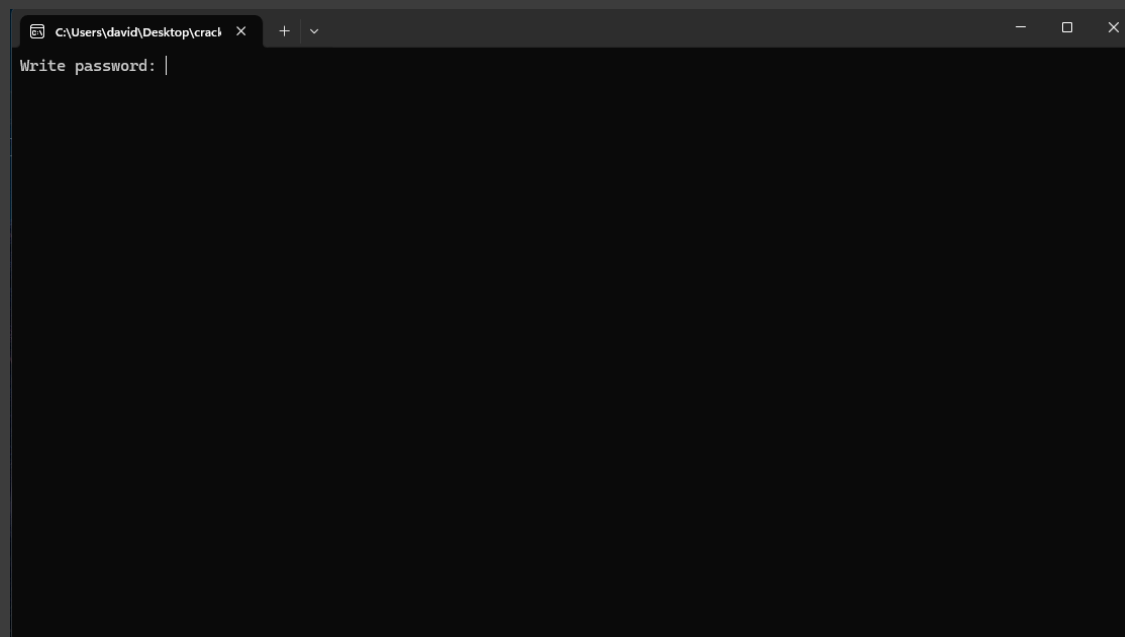
Here I can see the call to IsDebuggerPresent and then a conditional jump that checks if a debugger is present. If one is, it will skip the user input code path.

| | | | |
|--------------------|------------------|---|-------------------------------------|
| 00007FF68D071A04 | FF15 56680100 | call qword ptr ds:[<IsDebuggerPresent>] | |
| Breakpoint Enabled | 85C0 | test eax, eax | |
| 00007FF68D071A0A | 75 12 | jne decode.7FF68D071A20 | |
| 00007FF68D071A0E | 48:8D0D 04160100 | lea rcx, qword ptr ds:[7FF68D083019] | 00007FF68D083019:"write password: " |
| 00007FF68D071A15 | 48:83C4 28 | add rsp, 28 | |
| 00007FF68D071A19 | E9 C2F8FFFF | jmp decode.7FF68D0715E0 | |
| 00007FF68D071A1E | 66:90 | nop | |
| 00007FF68D071A20 | 48:8D0D EC150100 | lea rcx, qword ptr ds:[7FF68D083013] | 00007FF68D083013:"pause" |
| 00007FF68D071A27 | 48:83C4 28 | add rsp, 28 | |
| 00007FF68D071A2B | E9 90020100 | jmp <JMP.&system> | |

For the sake of convenience so I do not have to patch this check out on every execution restart within x64dbg, I patched the check out of the PE. This will streamline my reverse engineering of this CTF. *Note: when checking if I found the correct flag/key, I will use the **UNPATCHED** version. This patched version is just for dynamic analysis.*

| | | | |
|------------------|------------------|---|-------------------------------------|
| 00007FF68D071A04 | FF15 56680100 | call qword ptr ds:[<IsDebuggerPresent>] | |
| 00007FF68D071A0A | 90 | nop | |
| 00007FF68D071A0B | 90 | nop | |
| 00007FF68D071A0C | 90 | nop | |
| 00007FF68D071A0D | 90 | nop | |
| 00007FF68D071A0E | 48:8D0D 04160100 | lea rcx, qword ptr ds:[7FF68D083019] | 00007FF68D083019:"write password: " |

Simple test opening the new patched PE within x64dbg to see if I successfully patched out the anti-debugging code.





Now just to be extra cautious, I am going to add the breakpoint for kernel32.IsDebuggerPresent back in just to see if any other calls are made to it. No other hits, very nice! Now it is time that I can start looking, probing, and poking around.

After some stepping through some I land on what appears to be the main function.

| | |
|--|--|
| <pre> push rdi push rsi push rbx sub rsp,360 lea rsi,qword ptr ss:[rsp+20] lea rbx,qword ptr ss:[rsp+F0] call decode_debugger-check-patched-out.7FF648EC1B07 mov rdx,rsi lea rcx,qword ptr ds:[7FF648ED302A] lea rdi,qword ptr ss:[rsp+290] call decode_debugger-check-patched-out.7FF648EC16F0 mov rdx,rbx mov rcx,rsi lea rsi,qword ptr ss:[rsp+1C0] call decode_debugger-check-patched-out.7FF648EC1760 call decode_debugger-check-patched-out.7FF648EC1A00 mov rdx,rdi lea rcx,qword ptr ds:[7FF648ED3007] call decode_debugger-check-patched-out.7FF648EC1660 mov rdx,rsi mov rcx,rdi call decode_debugger-check-patched-out.7FF648EC1760 mov rdx,rsi mov rcx,rbx call decode_debugger-check-patched-out.7FF648EC1860 lea rcx,qword ptr ds:[7FF648ED303B] call decode_debugger-check-patched-out.7FF648EC15E0 lea rcx,qword ptr ds:[7FF648ED3013] call <JMP.&system> xor eax,eax add rsp,360 pop rbx pop rsi pop rdi ret </pre> | <pre> main function? 00007FF648ED302A:"Pa100-322-1L@101" output "Write password: " to console 00007FF648ED3007:"%" get user input 00007FF648ED3013:"pause" </pre> |
|--|--|

Playing around with it a bit it appears that it is indeed the Main program logic, where it is making calls to other functions for the handling of certain operations.

| | | |
|------------------|---|--|
| E8 55FCFEFF | call decode_debugger-check-patched-out.7FF64BEC1A00 | output "Write password: " to console |
| 48:89FA | mov rdx,rdi | rdi:"helloworld" |
| 48:8D0D 52120000 | lea rcx,qword ptr ds:[7FF64BED3007] | 00007FF64BED3007:"%" |
| E8 A6F8FEFF | call decode_debugger-check-patched-out.7FF64BEC1660 | get user input |
| 48:89F2 | mov rdx,rsi | |
| 48:89F9 | mov rcx,rdi | rdi:"helloworld" |
| E8 98F9FEFF | call decode_debugger-check-patched-out.7FF64BEC1760 | encoding function? |
| 48:89F2 | mov rdx,rsi | |
| 48:89D9 | mov rcx,rbx | rbx:"\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E" |
| E8 90FAFEFF | call decode_debugger-check-patched-out.7FF64BEC1860 | comparison function? |
| 48:8D0D 64120000 | lea rcx,qword ptr ds:[7FF64BED3038] | |
| E8 04F8FEFF | call decode_debugger-check-patched-out.7FF64BEC15E0 | |
| 48:8D0D 30120000 | lea rcx,qword ptr ds:[7FF64BED3013] | 00007FF64BED3013:"pause" |
| E8 D8FEFFFF | call <JMP.&system> | Press any key to continue... |
| 31C0 | xor eax,eax | |
| 48:81C4 60030000 | add rsp,360 | |
| 5B | pop rbx | rbx:"\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E" |
| 5E | pop rsi | |
| 5F | pop rdi | rdi:"helloworld" |
| C3 | ret | |

Right before it hits what looks like an encoding function, I notice that the *RBX* register is "`\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E`". This looks awfully a lot like a C-style escaped byte sequence stored as text. Lets go ahead and convert it back to plaintext: "`Pa>__-3\\->L@>_>`". Just for the heck of it, I will try to throw that in as the flag/key and see what the CTF responds with.

```

C:\Users\david\Desktop\crack x + v
Write password: Pa>__-3\\->L@>_>
Yes!
Press any key to continue . . .

```



Huh... to be completely honest I was not expecting that to be the flag/key...

Well, it seems that the flag/key is indeed "Pa>__-3||→L@>_>".

This felt too easy, so I challenged myself to breakdown and understand the assembly instructions for the encoding and comparing functions.

Lets go ahead and step into the encoding/comparison function to see if my original assumption on these functions was correct.

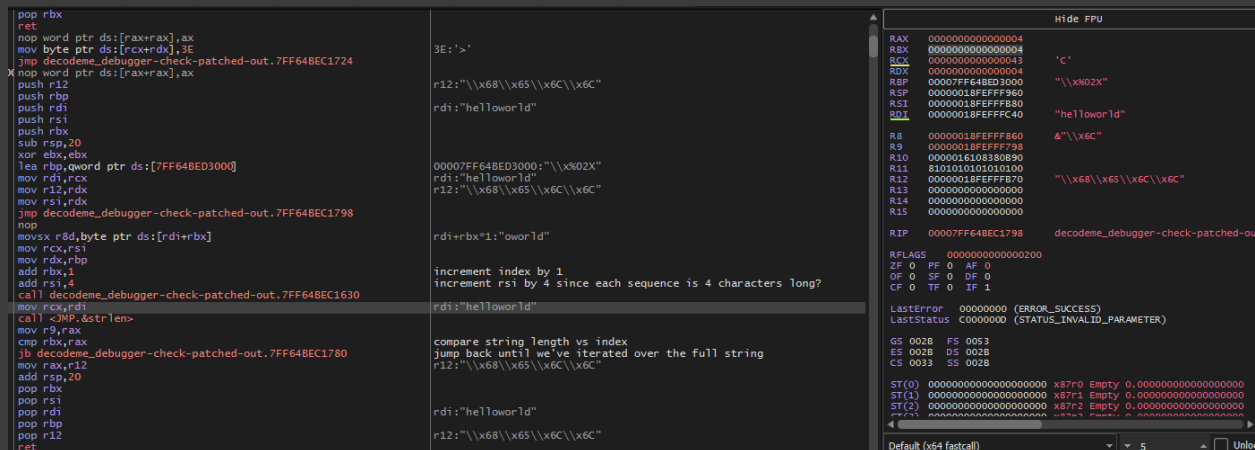
Encoding/transformation function:

```

41:54      push r12
55        push rbp
57        push rdi
56        push rsi
53        push rbx
48:83EC 20  sub rsp,20
31DB      xor ebx,ebx
48:8D2D 8D180100  lea rbp,qword ptr ds:[7FF64BED3000]
48:89CF      mov rdi,rcx
49:89D4      mov r12,rdx
48:89D6      mov rsi,rdx
EB 1A      jmp decodeme_debugger-check-patched-out.7FF64BEC1798
66:90      nop
44:0FB8E041F  movsx r8d,byte ptr ds:[rdi+rbx]
48:89F1      mov rcx,rsi
48:89EA      mov rdx,rbp
48:83C3 01    add rbx,1
48:83C6 04    add rsi,4
E8 98FEFFFF  call decodeme_debugger-check-patched-out.7FF64BEC1630
48:89F9      mov rcx,rdi
E8 00050100  call <JMP.&strlen>
49:89C1      mov r9,rax
48:39C3      cmp rbx,rax
72 D8      jb decodeme_debugger-check-patched-out.7FF64BEC1780
4C:89E0      mov rax,r12
48:83C4 20    add rsp,20
5B        pop rbx
5E        pop rsi
5F        pop rdi
5D        pop rbp
41:5C      pop r12
C3        ret

```

This function encodes each character of the user input into a C-style \xHH escape sequence (a specific kind of transformation/encoding). At the beginning of the function I noticed a string constant getting loaded into register *RBP*: "*\\x%02X*". Also, closer to the end of the function there is what appears to be a loop. It seems that this loop is iterating over the provided user input and using the value in *RBP* as a format string to convert each character into a C-style escaped byte sequence string. Then appending that output result onto the *R12* register. Which is then moved into the *RAX* (which is the return value of a function in Windows x64) register before the end of the function.



R12: "*\\x68\\x65\\x6C\\x6C\\x6F*" which is the C-style escaped byte sequence equivalent of "hello". So this confirms that this function is indeed the encoding function.

Here is an ASCII table for a quick reference:

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|-----|----|-----|-----------------------------|-----|----|-----|-------|-------|-----|----|-----|-------|-----|-----|----|-----|--------|-----|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 | | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | \$ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | (| (| 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 |) |) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [| [| 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 |] |] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | | DEL |

Source: www.LookupTables.com

After the encoding function, the program compares the encoded result against a constant buffer that contains the bytes corresponding to "Pa>__-3||→L@>_>". This is why entering exactly that string as the password satisfies the check and prints "Yes!".

Comparison function:

```

00007FF64BEC18F0 48:83EC 28      sub rsp,28
00007FF64BEC18F1 09:8A01      movzx eax,byte ptr ds:[rcx]
00007FF64BEC18F2 49:89C8      mov r8,rcx
00007FF64BEC18F3 49:89D1      mov r9,rdx
00007FF64BEC18F4 8B:00      test al,al
00007FF64BEC18F5 0F:84 83010000 je decodeme_debugger-check-patched-out.7FF64BEC19F8
00007FF64BEC18F6 48:1849 01      lea rcx,qword ptr ds:[rcx+1]
00007FF64BEC18F7 33:02      xor edx,edx
00007FF64BEC18F8 90      cmp al,5C
00007FF64BEC18F9 74 18      jne decodeme_debugger-check-patched-out.7FF64BEC189A
00007FF64BEC18FA 48:83C1 01      add rcx,1
00007FF64BEC18FB 41:8B0410      mov byte ptr ds:[r8+rdx],al
00007FF64BEC18FC 44:8D52 01      lea r10d,qword ptr ds:[rdx+1]
00007FF64BEC18FD 0F:84 8301 0F movzx eax,byte ptr ds:[rcx-1]
00007FF64BEC18FE 8B:00      test al,al
00007FF64BEC18FF 74 15      jne decodeme_debugger-check-patched-out.7FF64BEC18A8
00007FF64BEC1900 49:83D2      movsxd rdx,r10d
00007FF64BEC1901 3C 3C      cmp al,5C
00007FF64BEC1902 75 65      jne decodeme_debugger-check-patched-out.7FF64BEC187F
00007FF64BEC1903 0F:8B01      movzx eax,byte ptr ds:[rcx]
00007FF64BEC1904 48:83C1 01      add rcx,1
00007FF64BEC1905 8B:00      test al,al
00007FF64BEC1906 75 06      jne decodeme_debugger-check-patched-out.7FF64BEC187B
00007FF64BEC1907 41:8B02      mov r10d,edx
00007FF64BEC1908 4D:83D2      movsxd r10,r10d
00007FF64BEC1909 4D:01C2      add r10,r8
00007FF64BEC190A 41:8B0410      mov byte ptr ds:[r10],0
00007FF64BEC190B 41:0F8600      movzx eax,byte ptr ds:[r8]
00007FF64BEC190C 8B:00      test al,al
00007FF64BEC190D 0F:84 02010000 je decodeme_debugger-check-patched-out.7FF64BEC19C0
00007FF64BEC190E 49:8D48 01      lea rcx,qword ptr ds:[r8+1]
00007FF64BEC190F 33:02      xor edx,edx
00007FF64BEC1910 3C 78      cmp al,78
00007FF64BEC1911 74 18      jne decodeme_debugger-check-patched-out.7FF64BEC18E3
00007FF64BEC1912 48:83C1 01      add rcx,1
00007FF64BEC1913 41:8B0410      mov byte ptr ds:[r8+rdx],al
00007FF64BEC1914 44:8D52 01      lea r10d,qword ptr ds:[rdx+1]
00007FF64BEC1915 0F:84 8301 0F movzx eax,byte ptr ds:[rcx-1]
00007FF64BEC1916 8B:00      test al,al
00007FF64BEC1917 74 15      jne decodeme_debugger-check-patched-out.7FF64BEC18F1
00007FF64BEC1918 49:83D2      movsxd rdx,r10d
00007FF64BEC1919 3C 78      cmp al,78
00007FF64BEC191A 75 65      jne decodeme_debugger-check-patched-out.7FF64BEC18C8
00007FF64BEC191B 0F:8B01      movzx eax,byte ptr ds:[rcx]
00007FF64BEC191C 48:83C1 01      add rcx,1
00007FF64BEC191D 8B:00      test al,al
00007FF64BEC191E 75 06      jne decodeme_debugger-check-patched-out.7FF64BEC18C4
00007FF64BEC191F 41:8B02      mov r10d,edx
00007FF64BEC1920 4D:83D2      movsxd r10,r10d
00007FF64BEC1921 4D:01C2      add r10,r8
00007FF64BEC1922 41:8B0410      mov byte ptr ds:[r10],0
00007FF64BEC1923 41:0F8601      movzx eax,byte ptr ds:[r9]
00007FF64BEC1924 8B:00      test al,al
00007FF64BEC1925 0F:84 02010000 je decodeme_debugger-check-patched-out.7FF64BEC1908

```

Upon entering the function, we can see that the **RBX & RCX** registers are C-style escaped byte sequences of the flag/key "Pa>__-3||→L@>_>" and **RAX & RDX** registers are the same but of my user input "helloworld". What else stands out on first glance is how long this comparison function is too. At the start of the function I notice that the function execution is skipped if the string length is 0.

| | |
|--|--|
| <pre> sub rsp,28 movzx eax,byte ptr ds:[rcx] mov r8,rcx mov r9,rdx test al,al je decodeme_debugger-check-patched-out.7FF64BEC19F8 </pre> | <pre> r8: "x50x61x3Ex5Fx5Fx2Dx33x5Cx5Cx2Dx3Ex4Cx40x3Ex5Fx3E" r9: "\\x68\\x65\\x6C\\x6F\\x77\\x6F\\x72\\x6C\\x64" skip function execution if string is empty </pre> |
|--|--|

It seems that it enters a loop where it begins decoding the flag/key parsing the C-style \x.. escape string and normalizing it into a "clean" hex representation in another buffer. So "\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E" gets transformed into "x50x61x3Ex5Fx5Fx2Dx33x5Cx5Cx2Dx3Ex4Cx40x3Ex5Fx3E".

| | |
|---|--|
| <pre> lea rcx,qword ptr ds:[rcx+1] xor edx,edx cmp al,5C je decodeme_debugger-check-patched-out.7FF64BEC189A add rcx,1 mov byte ptr ds:[r8+rdx],al lea r10d,qword ptr ds:[rdx+1] movzx eax,byte ptr ds:[rcx-1] test al,al je decodeme_debugger-check-patched-out.7FF64BEC18A8 movsxd rdx,r10d cmp al,5C jne decodeme_debugger-check-patched-out.7FF64BEC187F movzx eax,byte ptr ds:[rcx] add rcx,1 test al,al jne decodeme_debugger-check-patched-out.7FF64BEC187B </pre> | <pre> move source pointer forward by one zero EDX (loop index) compare EAX against character '\' (5C) jump over main loop logic if character is '\' (5C) advance source pointer by 1 write the character to the destination buffer increment destination buffer index reload last character check if character is string terminator (al == 0) jump to loop exit if it is move destination buffer index back into RDX compare EAX against character '\' (5C) again grab the next character since it is assumed to be hex bytes </pre> |
| <pre> add rcx,1 test al,al jne decodeme_debugger-check-patched-out.7FF64BEC187B </pre> | <pre> advance source pointer by 1 again check if character is string terminator (al == 0) last loop instruction: jump to start if character is not string terminator </pre> |

After another safety check to see if the string length of the output is 0, the execution continues into another near identical loop that now replaces the "x" (0x78) character instead of the '\ (0x5C) character.

So "x50x61x3Ex5Fx5Fx2Dx33x5Cx5Cx2Dx3Ex4Cx40x3Ex5Fx3E" gets transformed into "50613E5F5F2D335C5C2D3E4C403E5F3E".

```
lea rcx,qword ptr ds:[r8+1]
xor edx,edx
cmp al,78
je decode_me_debugger-check-patched-out.7FF648EC18E3
add rcx,1
mov byte ptr ds:[r8+rdx],al
lea r10d,qword ptr ds:[rdx+1]
movzx eax,byte ptr ds:[rcx-1]
test al,al
je decode_me_debugger-check-patched-out.7FF648EC18F1
movsxd rdx,r10d
cmp al,78
jne decode_me_debugger-check-patched-out.7FF648EC18C8
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decode_me_debugger-check-patched-out.7FF648EC18C4
```

And another safety check for the output string length, execution is then followed by two identical loops as the previous two. This time operating on the user input - "helloworld".

```
lea rcx,qword ptr ds:[r9+1]
xor edx,edx
cmp al,5C
je decode_me_debugger-check-patched-out.7FF648EC192C
add rcx,1
mov byte ptr ds:[r9+rdx],al
lea r10d,qword ptr ds:[rdx+1]
movzx eax,byte ptr ds:[rcx-1]
test al,al
je decode_me_debugger-check-patched-out.7FF648EC193A
movsxd rdx,r10d
cmp al,5C
jne decode_me_debugger-check-patched-out.7FF648EC1911
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decode_me_debugger-check-patched-out.7FF648EC190D
mov r10d,edx
movsxd r10,r10d
add r10,r9
mov byte ptr ds:[r10],0
movzx eax,byte ptr ds:[r9]
test al,al
je decode_me_debugger-check-patched-out.7FF648EC19F0
lea rcx,qword ptr ds:[r9+1]
xor edx,edx
cmp al,78
je decode_me_debugger-check-patched-out.7FF648EC1975
add rcx,1
mov byte ptr ds:[r9+rdx],al
lea r10d,qword ptr ds:[rdx+1]
movzx eax,byte ptr ds:[rcx-1]
test al,al
je decode_me_debugger-check-patched-out.7FF648EC1983
movsxd rdx,r10d
cmp al,78
jne decode_me_debugger-check-patched-out.7FF648EC195A
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decode_me_debugger-check-patched-out.7FF648EC1956
```

So "\\x68\\x65\\x6C\\x6C\\x6F\\x77\\x6F\\x72\\x6C\\x64" becomes "x68x65x6Cx6Cx6Fx77x6Fx72x6Cx64", which becomes "68656C6C6F776F726C64".

Finally, here I see it loading the two output buffers for the newly transformed flag/key into register *RDX* as well as the transformed user input into register *RCX*. Which is followed by a call to *strcmp*.

On Windows x64, the first two function arguments that are passed in: *RCX* as argument #1 and *RDX* are argument #2.

```
mov rdx,r9
mov rcx,r8
call <JMP.&strcmp>
test eax,eax
jne decode_me_debugger-check-patched-out.7FF64BEC19B0
```

If the strings match, then the *test* instruction will return 0 which will set the ZF (zero flag) to 1. Since *strcmp* returns zero if the strings match, and non-zero if they differ.

If the strings do **NOT** match (ZF == 0) then we output "No!", otherwise the strings match (ZF == 1) then output "Yes!".

| | |
|---|---|
| <pre>mov rdx,r9 mov rcx,r8 call <JMP.&strcmp> test eax,eax jne decode_me_debugger-check-patched-out.7FF64BEC19B0 lea rcx,qword ptr ds:[7FF64BED300A] add rsp,28 jmp decode_me_debugger-check-patched-out.7FF64BEC15E0 nop dword ptr ds:[rax],eax lea rcx,qword ptr ds:[7FF64BED300F] add rsp,28 jmp decode_me_debugger-check-patched-out.7FF64BEC15E0</pre> | <p>rdx:"68656C6C6F776F726C64", r9:"68656C6C6F776F726C64" rcx:"50613E5F5F2D335C5C2D3E4C403E5F3E", r8:"50613E5F"</p> <p>if strings do not match then jump load "Yes!" into memory and print it to console</p> <p>load "No!" into memory and print it to console</p> |
|---|---|

So, in result the flag/key for this CTF is: "Pa>_-3||→L@>_>".

Final thoughts:

Upon second analysis of the string references within the CTF PE module I noticed two references to the string formatter constant used.

| Address | Disassembly | String Address | String |
|------------------|-------------------------------------|------------------|-----------|
| 00007FF64BEC1634 | lea rdx,qword ptr ds:[7FF64BED3000] | 00007FF64BEC3000 | "\\x%02X" |
| 00007FF64BEC1689 | lea rdx,qword ptr ds:[7FF64BED3007] | 00007FF64BEC3007 | "%S" |
| 00007FF64BEC176C | lea rbp,qword ptr ds:[7FF64BED3000] | 00007FF64BEC3000 | "\\x%02X" |

This could have been a big hint given away on the type of encoding/transformation that is being used.

I am also glad I decided to do a deep dive into the encoding and comparing function as that helped me gain a lot of insight and deepen my understanding of current and new knowledge. It also helped familiarize some concepts better as well as engrain them into my brain.