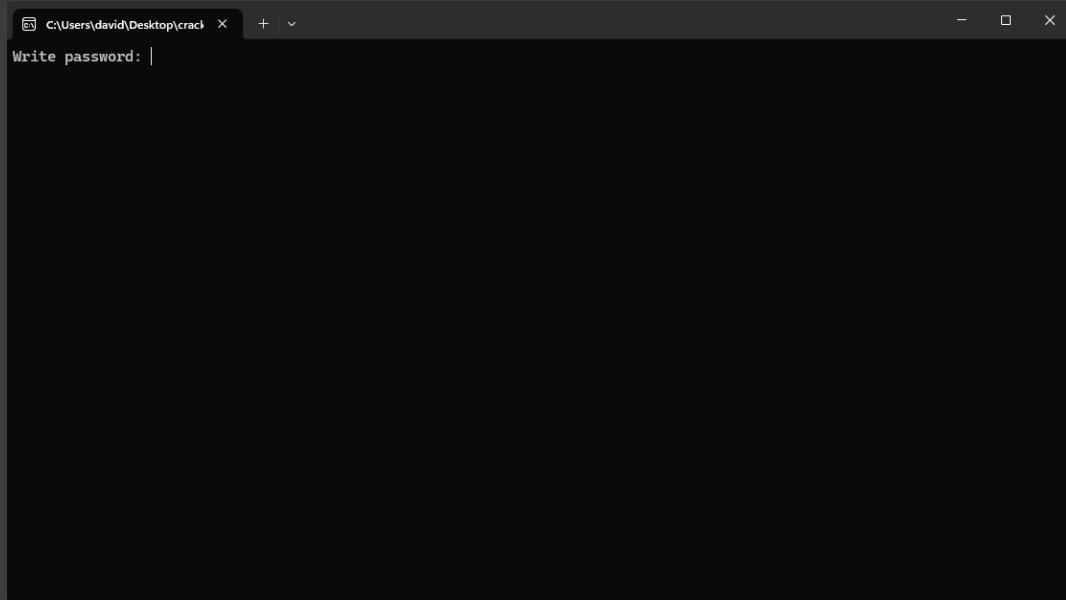


Crackme: <https://crackmes.one/crackme/69245c422d267f28f69b806e>

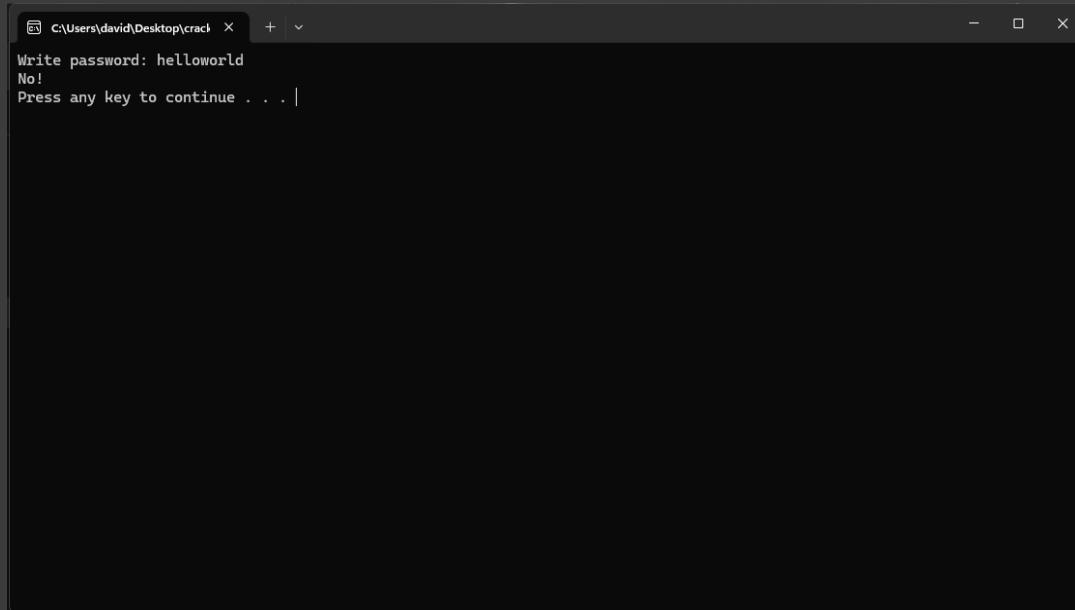
Author: vilxd Software Used: x64dbg Writeup By: SenorGPT

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

Regular execution (startup):



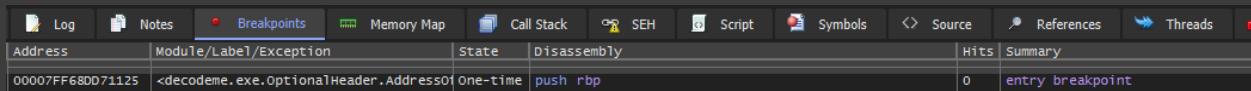
Wrong answer provided - proceeded by termination on following key press:



**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.



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.

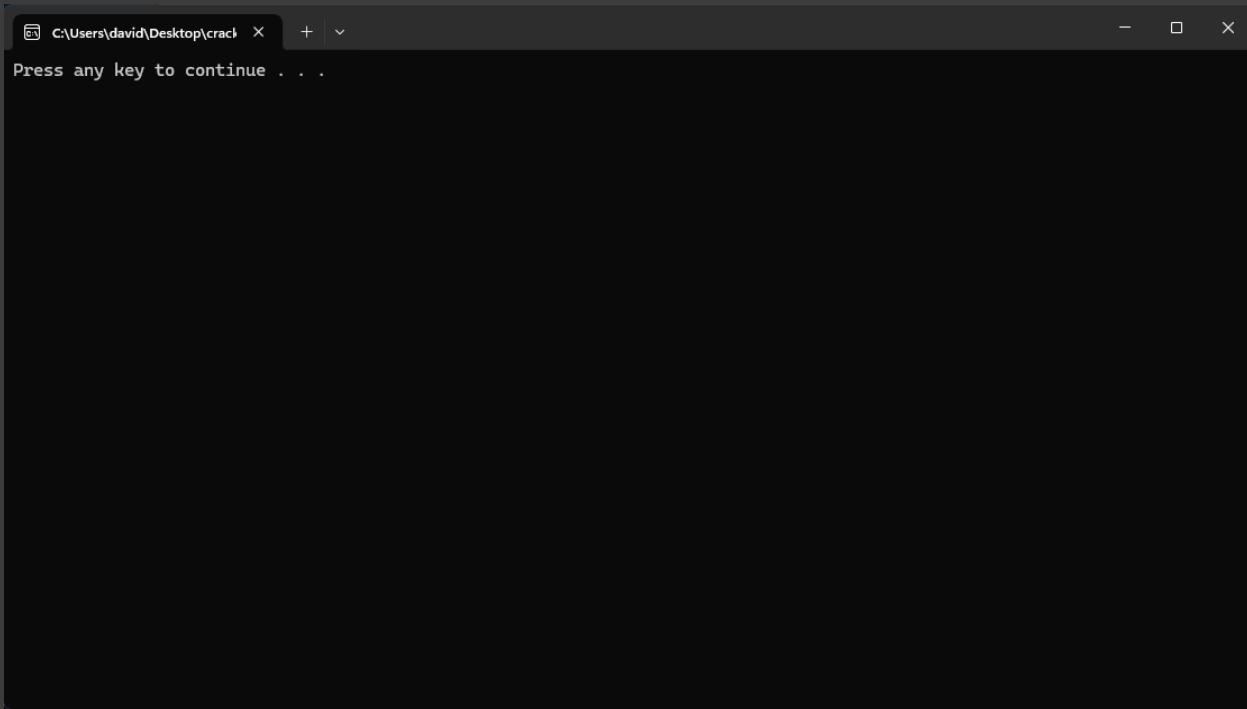
Address	Disassembly	String Address	String
00007FF68D071634	lea rdx,qword ptr ds:[7FF68DD83000]	00007FF68DD83000	"\\x00Z"
00007FF68D071689	lea rdx,qword ptr ds:[7FF68DD83007]	00007FF68DD83007	"\\x00Z"
00007FF68D07169C	lea rdx,qword ptr ds:[7FF68DD83007]	00007FF68DD83007	"\\x00Z"
00007FF68D07139C	lea rdx,qword ptr ds:[7FF68DD8300A]	00007FF68DD8300A	"Yes!"
00007FF68D071380	lea rcx,qword ptr ds:[7FF68DD8300C]	00007FF68DD8300F	"No!"
00007FF68D071A0E	lea rcx,qword ptr ds:[7FF68DD83019]	00007FF68DD83019	"Write password: "
00007FF68D071A20	lea rcx,qword ptr ds:[7FF68DD83013]	00007FF68DD83013	"pause"
00007FF68D071C9C	lea rax,qword ptr ds:[7FF68DD83080]	00007FF68DD83080	"Argument domain error (DOMAIN)"
00007FF68D071C9A9	lea rax,qword ptr ds:[7FF68DD83080]	00007FF68DD83080	"Argument signature error (SIGNATURE)"
00007FF68D071C168	lea rax,qword ptr ds:[7FF68DD830C0]	00007FF68DD830C0	"VirtualAlloc() range error (OVERFLOW)"
00007FF68D071C1C3	lea rax,qword ptr ds:[7FF68DD830E0]	00007FF68DD830E0	"Partial loss of significance (PLOSS)"
00007FF68D071C1D0	lea rax,qword ptr ds:[7FF68DD83108]	00007FF68DD83108	"Total loss of significance (TLOSS)"
00007FF68D071C1D0	lea rax,qword ptr ds:[7FF68DD83130]	00007FF68DD83130	"The result is too small to be represented (UNDERFLOW)"
00007FF68D071CEA	lea rax,qword ptr ds:[7FF68DD83166]	00007FF68DD83166	"Unknown error"
00007FF68D071C944	lea rax,qword ptr ds:[7FF68DD83166]	00007FF68DD83166	"matherr(0) error in %g(%g, %g) (retval=%g)\n"
00007FF68D071D0B8	lea rax,qword ptr ds:[7FF68DD831C0]	00007FF68DD831C0	"VirtualAlloc() runtime failure: %p\n"
00007FF68D071E87	lea rax,qword ptr ds:[7FF68DD831E0]	00007FF68DD831E0	"Address %p has no image-section"
00007FF68D071FB4	lea rax,qword ptr ds:[7FF68DD83200]	00007FF68DD83200	"VirtualQuery failed for %d bytes at address %p"
00007FF68D0720A8	lea rax,qword ptr ds:[7FF68DD83238]	00007FF68DD83238	"VirtualProtect failed with code 0x00000000"
00007FF68D072200	lea rax,qword ptr ds:[7FF68DD83260]	00007FF68DD83260	"Unknown pseudo relocation protocol version %d.\n"
00007FF68D07240C	lea rax,qword ptr ds:[7FF68DD83298]	00007FF68DD83298	"Unknown pseudo relocation bit size %d.\n"
00007FF68D07240C	lea rax,qword ptr ds:[7FF68DD83298]	00007FF68DD83298	"%d-bit pseudo relocation at %p out of range, targeting %p, yielding the value %p.\n"
00007FF68D075AC5	lea r9,qword ptr ds:[7FF68DD83350]	00007FF68DD83350	"(nil)"
00007FF68D07605E	lea rax,qword ptr ds:[7FF68DD83356]	00007FF68DD83356	"nan"
00007FF68D0761A0	lea rax,qword ptr ds:[7FF68DD8335A]	00007FF68DD8335A	"inf"
00007FF68D07630C	lea rax,qword ptr ds:[7FF68DD8335E]	00007FF68DD8335E	"infty"
00007FF68D0763A3	lea rax,qword ptr ds:[7FF68DD83360]	00007FF68DD83360	"(nil)"
00007FF68D07654F	lea rax,qword ptr ds:[7FF68DD83364]	00007FF68DD83364	"NaN"
00007FF68D079A20	lea rdx,qword ptr ds:[7FF68DD83626]	00007FF68DD83626	"NaN"
00007FF68D079A7B	lea rdx,qword ptr ds:[7FF68DD8362A]	00007FF68DD8362A	"Inf"
00007FF68D079B2F	lea rdx,qword ptr ds:[7FF68DD83626]	00007FF68DD83626	"NaN"
00007FF68D079B86	lea rdx,qword ptr ds:[7FF68DD8362A]	00007FF68DD8362A	"Inf"
00007FF68D07ABFD	lea rdx,qword ptr ds:[7FF68DD837A0]	00007FF68DD837A0	"Infinity"
00007FF68D07A8E0	lea rdx,qword ptr ds:[7FF68DD837A4]	00007FF68DD837A4	"IN"
00007FF68D07E231	lea rdx,qword ptr ds:[7FF68DD837D7]	00007FF68DD837D7	"nf"
00007FF68D07E258	lea rdx,qword ptr ds:[7FF68DD837D7]	00007FF68DD837D7	"infinity"
00007FF68D07E287	lea rdx,qword ptr ds:[7FF68DD833E5]	00007FF68DD833E5	"an"
00007FF68D080AE1	lea rax,qword ptr ds:[7FF68DD834E0]	00007FF68DD834E0	"0123456789"
00007FF68D080AE1	lea rax,qword ptr ds:[7FF68DD834E0]	00007FF68DD834E0	"0123456789"
00007FF68D080B1F	lea rax,qword ptr ds:[7FF68DD834F7]	00007FF68DD834F7	"ABCDDEF"
00007FF68D080B1F	lea rax,qword ptr ds:[7FF68DD834F7]	00007FF68DD834F7	"ABCDDEF"
00007FF68D08317F	lea rdx,qword ptr ds:[7FF68DD8302A]	00007FF68DD8302A	"Pa100-322-1L@101"
00007FF68D0831AE	lea rdx,qword ptr ds:[7FF68DD83007]	00007FF68DD83007	"%s"
00007FF68D0831DC	lea rdx,qword ptr ds:[7FF68DD83013]	00007FF68DD83013	"pause"
00007FF68D08322D	and byte ptr ds:[7FF68DD832A],ah	00007FF68DD832A3	"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	decodem.e.00007FF68DD71A0A
000000E2C7FFF788	00007FF68DD71340	00007FF68DD81DAB	80	User	decodem.e.00007FF68DD81DAB
000000E2C7FFF808	00007FF68DD71146	00007FF68DD71340	40	User	decodem.e.00007FF68DD71340
000000E2C7FFF848	00007FFFC16EE8D7	00007FF68DD71146	30	System	decodem.e.00007FF68DD71146
000000E2C7FFF878	00007FFFC1EAC53C	00007FFFC16EE8D7	50	System	kernel32.BaseThreadInitThunk+17
000000E2C7FFF8C8	000000000000000000	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.

● 00007FF68DD71A04	FF15 56680100	call qword ptr ds:[<IsDebuggerPresent>]	
Breakpoint Enabled A0C	85C0	test eax,eax	
✓ 00007FF68DD71A0E	75 12	jne decodeme_7FF68DD71A20	00007FF68DD83019;"Write password: "
00007FF68DD71A0F	48:80D0 04160100	lea rcx,qword ptr ds:[<FF68DD83019>]	
00007FF68DD71A15	48:83C4 28	add rsp,28	
00007FF68DD71A19	E9 0000FFFF	jmp decodeme_7FF68DD715E0	
00007FF68DD71A1E	66:50	nop	
00007FF68DD71A20	48:80D0 EC150100	lea rcx,qword ptr ds:[<FF68DD83013>]	00007FF68DD83013;"pause"
00007FF68DD71A27	48:83C4 28	add rsp,28	
00007FF68DD71A2E	E9 90020100	jmp <JMP.&system>	JMP.&_C_specific_handler
00007FF68DD71A30	FF25 72680100	jmp qword ptr ds:[<_C_specific_handler>]	
00007FF68DD71A36	90	nop	
00007FF68DD71A37	90	nop	
00007FF68DD71A3B	FF25 42680100	jmp qword ptr ds:[<Sleep>]	JMP.&Sleep
00007FF68DD71A3C	90	nop	
00007FF68DD71A3F	90	nop	
00007FF68DD71A40	FF25 32680100	jmp qword ptr ds:[<SetUnhandledExceptionFilter>]	JMP.&SetUnhandledExceptionFilter
00007FF68DD71A46	90	nop	
00007FF68DD71A47	90	nop	
00007FF68DD71A48	FF25 12680100	jmp qword ptr ds:[<IsDebuggerPresent>]	JMP.&IsDebuggerPresent
00007FF68DD71A49	90	nop	
00007FF68DD71A4F	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.

00007FF68DD83019	FF15 56680100	call qword ptr ds:[<IsDebuggerPresent>]	00007FF68DD83019:"Write password: "
00007FF68DD01A0A	85C0	test eax,eax	
00007FF68DD01A0C	75 12	je decodeme,7FF68DD01A20	
00007FF68DD01A0E	48:8D00 04160100	lea rcx,qword ptr ds:[7FF68DD01A09]	
00007FF68DD01A15	48:83C4 28	add rsp,28	
00007FF68DD01A19	E9 C2FBFFFF	jmp decodeme,7FF68DD015E0	
00007FF68DD01A1E	66:90	nop	
00007FF68DD01A20	48:8D00 EC150100	lea rcx,qword ptr ds:[7FF68DD01A03]	00007FF68DD83013:"pause"
00007FF68DD01A27	48:83C4 28	add rsp,28	
00007FF68DD01A2B	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.*

```
● 00007FF68DD71A04 FF15 56680100 call qword ptr ds:[IsDebuggerPresent]
] 00007FF68DD71A04 90 nop
] 00007FF68DD71A08 90 nop
] 00007FF68DD71A0C 90 nop
] 00007FF68DD71A0D 90 nop
] 00007FF68DD71A0E 48:80D0 04160100 lea rax,qword ptr ds:[_FF68DD83019]
00007FF68DD83019:"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.

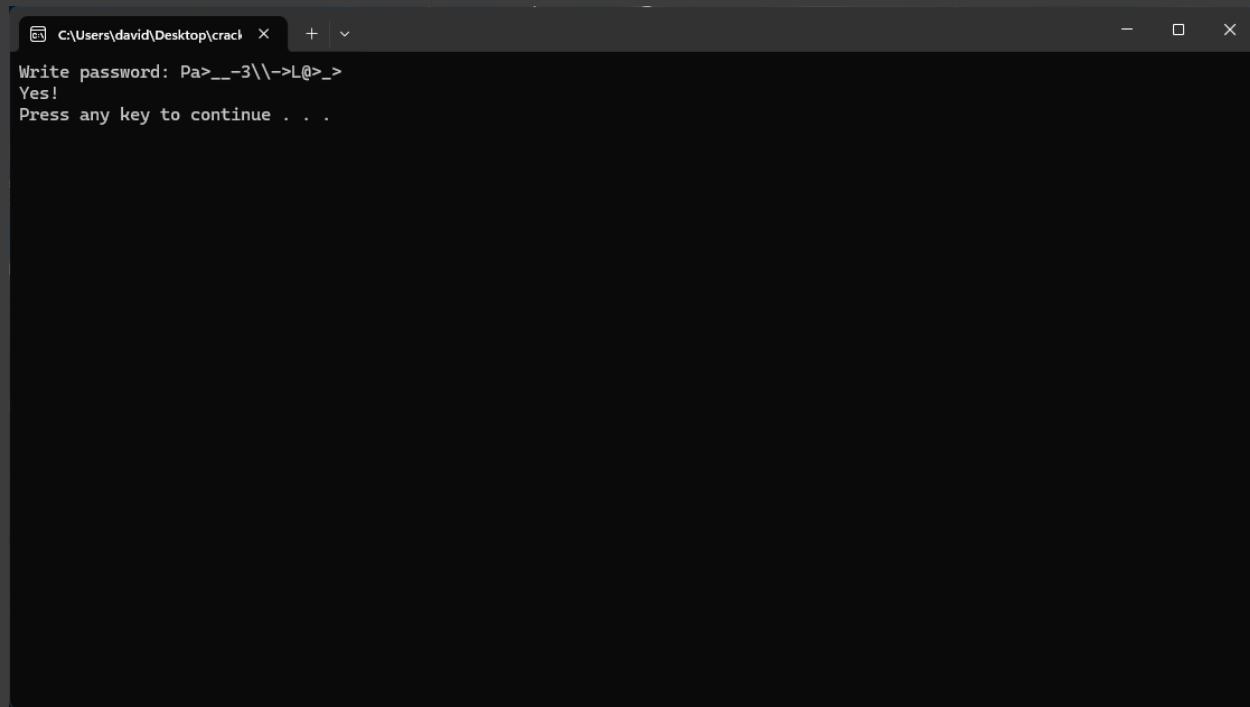
```
push rdi
push rsi
push rbx
sub rsp,360
lea rsi,qword ptr ss:[rsp+20]
lea rbx,qword ptr ss:[rsp+F0]
call decodeme_debugger-check-patched-out.7FF64BEC1B07
mov rdx,rsi
lea rcx,qword ptr ds:[7FF64BED302A]
lea rdi,qword ptr ss:[rsp+290]
call decodeme_debugger-check-patched-out.7FF64BEC16F0
mov rdx,rbx
mov rcx,rsi
lea rsi,qword ptr ss:[rsp+1C0]
call decodeme_debugger-check-patched-out.7FF64BEC1760
call decodeme_debugger-check-patched-out.7FF64BEC1A00
mov rdx,rdi
lea rcx,qword ptr ds:[7FF64BED3007]
call decodeme_debugger-check-patched-out.7FF64BEC1660
mov rdx,rsi
mov rcx,rdi
call decodeme_debugger-check-patched-out.7FF64BEC1760
mov rdx,rsi
mov rcx,rbx
call decodeme_debugger-check-patched-out.7FF64BEC1860
lea rcx,qword ptr ds:[7FF64BED3038]
call decodeme_debugger-check-patched-out.7FF64BEC15E0
lea rcx,qword ptr ds:[7FF64BED3013]
call <JMP.&system>
xor eax,eax
add rsp,360
pop rbx
pop rsi
pop rdi
ret
```

main function?  
00007FF64BED302A:"Pa100-322-1L@101"  
output "Write password: " to console  
00007FF64BED3007:"%s"  
get user input  
00007FF64BED3013:"pause"

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 55FCFFFF	call decodeme_debugger-check-patched-out.7FF64BEC1A00	output "Write password: " to console
48:89FA	mov rdx,rdi	rdi:"helloworld"
48:800D 52120000	lea rcx,qword ptr ds:[7FF64BED3007]	00007FF64BED3007:"%"
E8 A6F8EFFF	call decodeme_debugger-check-patched-out.7FF64BEC1660	get user input
48:89F2	mov rdx,r8	
48:89F9	mov rcx,rdi	rdi:"helloworld"
E8 9BF9EFFF	call decodeme_debugger-check-patched-out.7FF64BEC1760	encoding function?
48:89F2	mov rdx,r8	rbx:"\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E"
48:89D9	mov rcx,rbx	comparison function?
E8 90FAEFFF	call decodeme_debugger-check-patched-out.7FF64BEC1860	00007FF64BED3013:"pause"
48:800D 64120000	lea rcx,qword ptr ds:[7FF64BED3038]	Press any key to continue...
E8 04F8EFFF	call decodeme_debugger-check-patched-out.7FF64BEC15E0	
48:800D 30120000	lea rcx,qword ptr ds:[7FF64BED3013]	
E8 D8FEFFF	call <IMP.&system>	rbx:"\\x50\\x61\\x3E\\x5F\\x5F\\x2D\\x33\\x5C\\x5C\\x2D\\x3E\\x4C\\x40\\x3E\\x5F\\x3E"
31C0	xor eax,eax	rdi:"helloworld"
48:81C4 60030000	add rsp,360	
5B	pop rbx	
5E	pop r8	
5F	pop rdi	
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.





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
44:0FBE041F    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
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.

```

pop rbx
ret
nop word ptr ds:[rax+rax],ax
nop byte ptr ds:[rcx+rdx],3E
jmp decodeme_debugger-check-patched-out.7FF64BEC1724
nop word ptr ds:[r12+rax],ax
push r12
push rbp
push rdi
push rsi
push rbx
sub rsp,20
xor ebx,ebx
lea rbp,qword ptr ds:[7FF64BED3000]
mov rdi,rcx
mov rdx,rsp
mov rsi,r12
mov r12,rdx
jmp decodeme_debugger-check-patched-out.7FF64BEC1798
nop
movsx r8d,byte ptr ds:[rdi+rbx]
mov rcx,rsi
mov rdx,rsp
add rbx,1
add rsi,4
call decodeme_debugger-check-patched-out.7FF64BEC1630
mov rcx,rdi
call rbp,&strlen>
mov r9,rax
cmp rbx,rcx
jb decodeme_debugger-check-patched-out.7FF64BEC1780
mov rax,r12
add rbp,20
pop rsi
pop rdi
pop rbp
pop r12
ret

```

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

Source: [www.LookupTables.com](http://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:

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.

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".

```
lea rcx,qword ptr ds:[rcx+1]
xor edx,edx
cmp al,5C
je decodememe_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 decodememe_debugger-check-patched-out.7FF64BEC18A8
movsx rdx,r10d
cmp al,5C
jne decodememe_debugger-check-patched-out.7FF64BEC187F
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decodememe_debugger-check-patched-out.7FF64BEC187B
```

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  
  
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

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 "x50×61×3Ex5Fx5Fx2Dx33×5Cx5Cx2Dx3Ex4Cx40×3Ex5Fx3E" gets transformed into "50613E5F5F2D335C5C2D3E4C403E5F3E".

```
lea rcx,qword ptr ds:[r8+1]
xor edx,edx
cmp al,78
je decodeme_debugger-check-patched-out.7FF64BEC18E3
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.7FF64BEC18F1
movsx rdx,r10d
cmp al,78
jne decodeme_debugger-check-patched-out.7FF64BEC18C8
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decodeme_debugger-check-patched-out.7FF64BEC18C4
...
```

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 decodeme_debugger-check-patched-out.7FF64BEC192C
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 decodeme_debugger-check-patched-out.7FF64BEC193A
movsx rdx,r10d
cmp al,5C
jne decodeme_debugger-check-patched-out.7FF64BEC1911
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
je decodeme_debugger-check-patched-out.7FF64BEC190D
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 decodeme_debugger-check-patched-out.7FF64BEC19F0
lea rcx,qword ptr ds:[r9+1]
xor edx,edx
cmp al,78
je decodeme_debugger-check-patched-out.7FF64BEC1975
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 decodeme_debugger-check-patched-out.7FF64BEC1983
movsx rdx,r10d
cmp al,78
jne decodeme_debugger-check-patched-out.7FF64BEC195A
movzx eax,byte ptr ds:[rcx]
add rcx,1
test al,al
jne decodeme_debugger-check-patched-out.7FF64BEC1956
...
```

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

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]	00007FF64BED3000	"\\x%02X"
00007FF64BEC1689	lea rcx,qword ptr ds:[7FF64BED3007]	00007FF64BED3007	"%s"
00007FF64BEC176C	lea rbp,qword ptr ds:[7FF64BED3000]	00007FF64BED3000	"\\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.