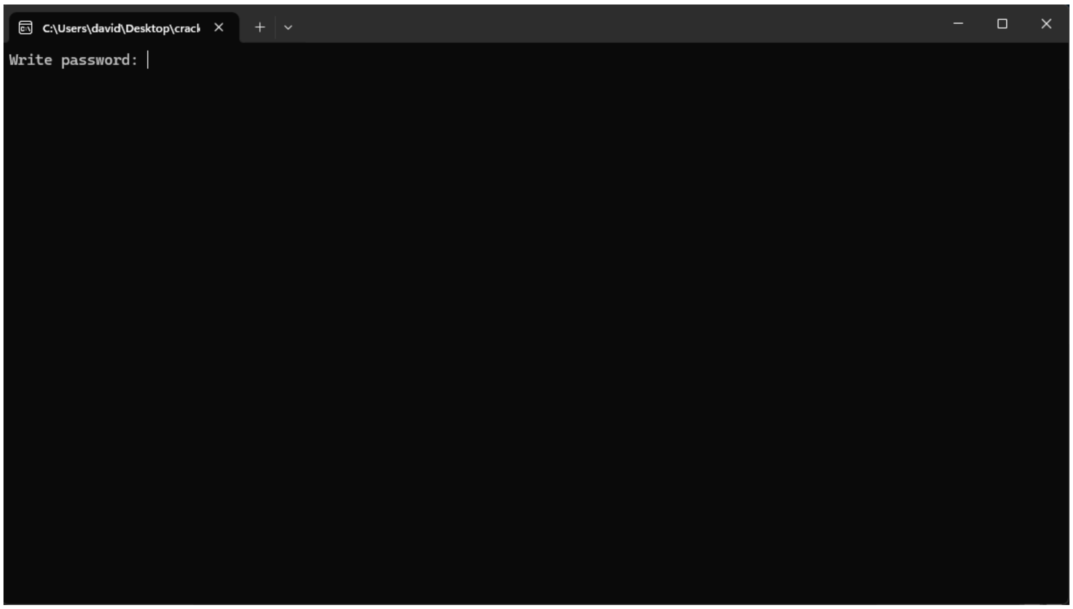
Crackme: https://crackmes.one/crackme/69245c422d267f28f69b806e

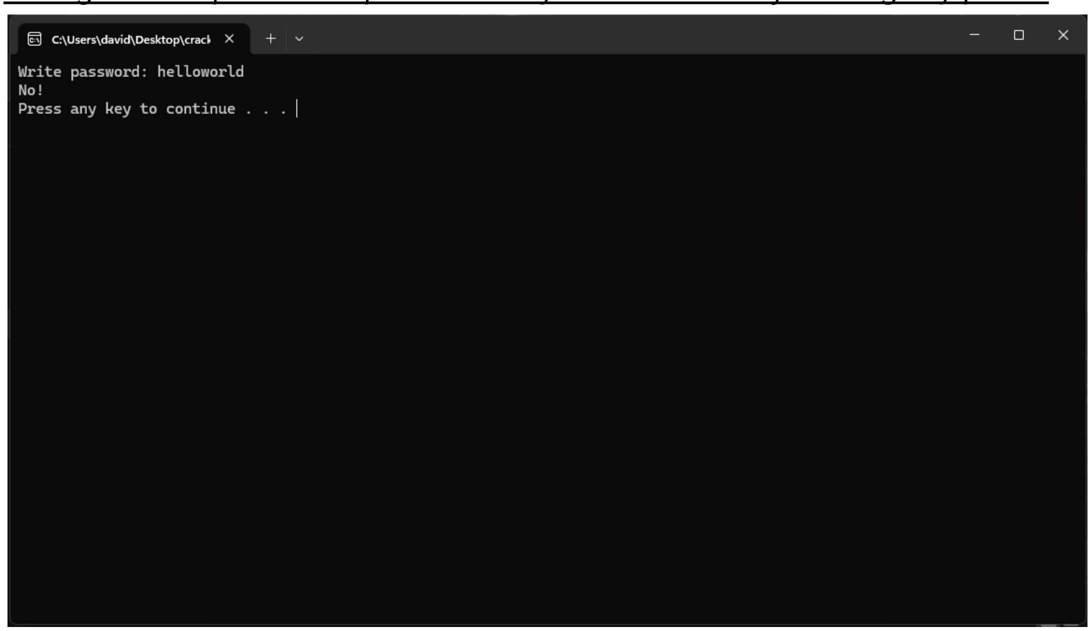
Author: vilxd Software Used: x64dbg Writeup By: SenorGPT

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

Regular execution (startup):



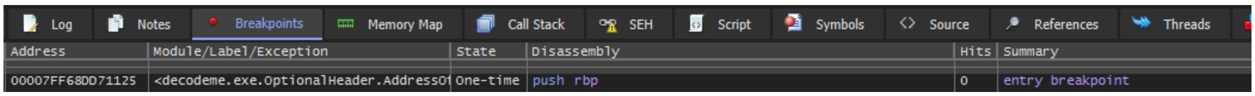
Wrong answer provided - proceeded by termination on following key press:



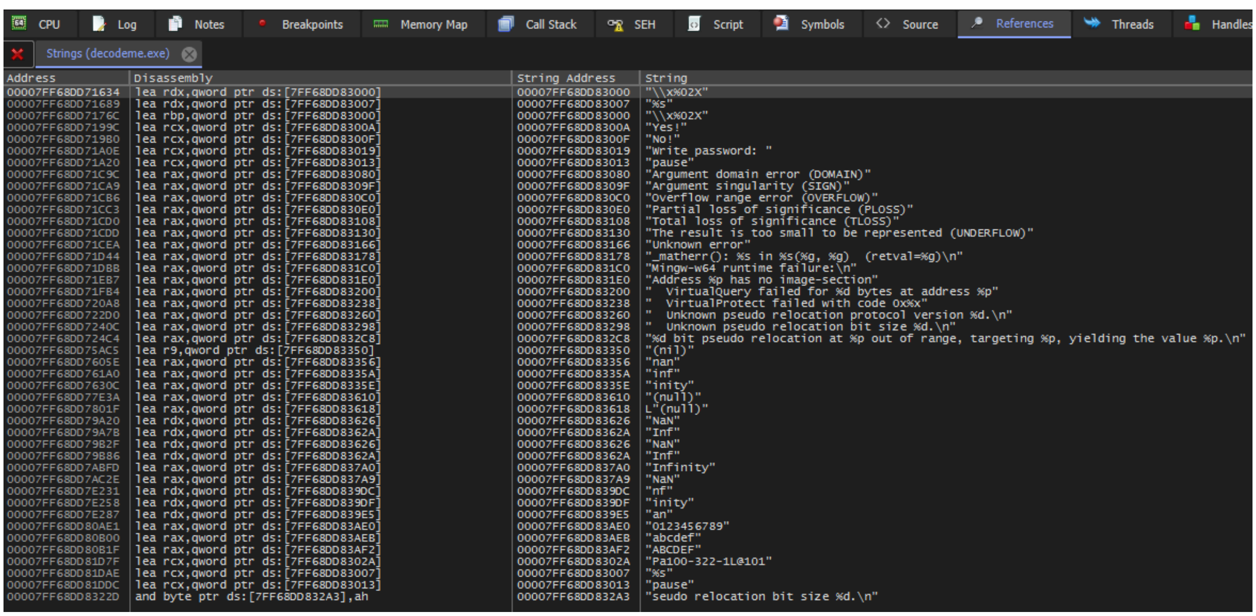
Goal: recover the correct password string (no patching required to solve), using static and/or dynamic analysis.

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.

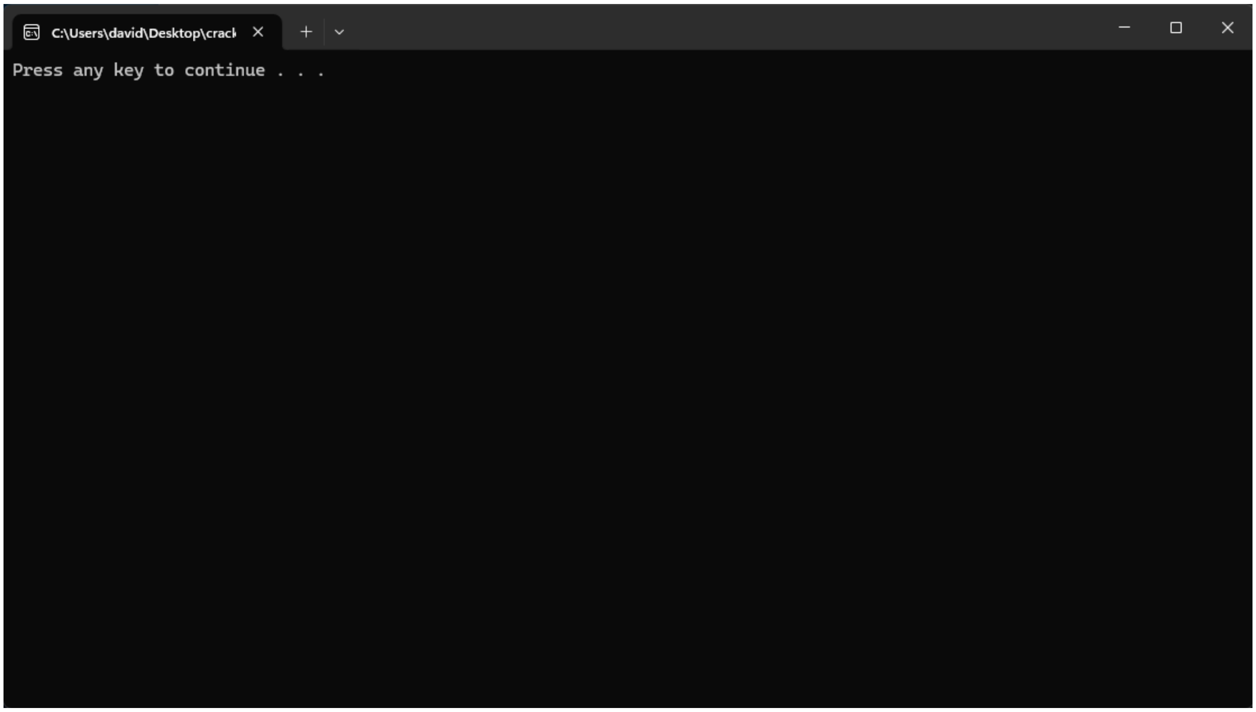


Then going into Symbols and double clicking the CTF PE (decodeme.exe) to bring me into the CPU tab with the disassembly view. Since I am now in the target module, I can right click the disassembly empty space — Search For — Current Module — String References.



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!
```

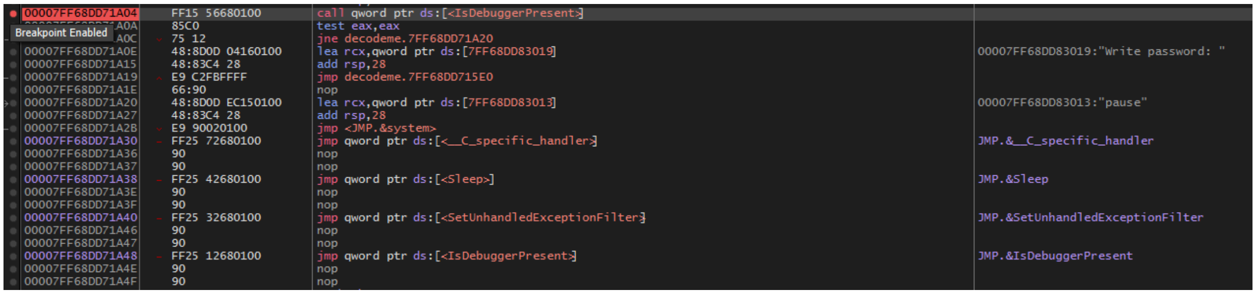
Once added, I restart program execution. I got a few hits on ntdll.NtQueryInformationProcess, but nothing interesting.
Then, I got a break on kernel32.IsDebuggerPresent. Things are starting to get spicy.

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.

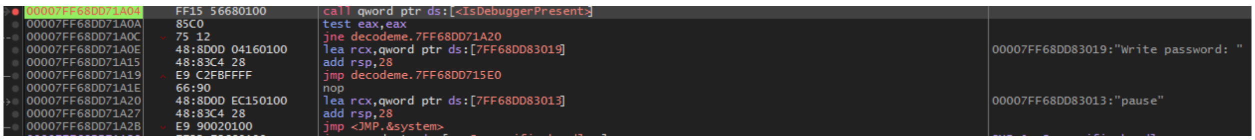
Breakpoints | Memory Map | Call Stack | SEH | Script | Symbols | Source | Reference

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

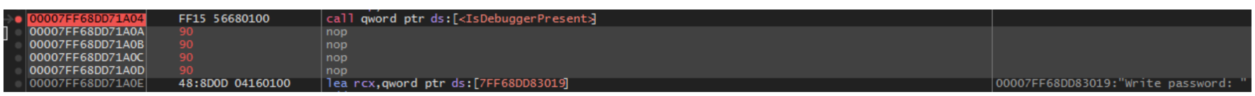
Double clicking on the frame underneath the kernel32.IsDebuggerPresent call brings me to the function that called it, which appears to be the main logic of the program.



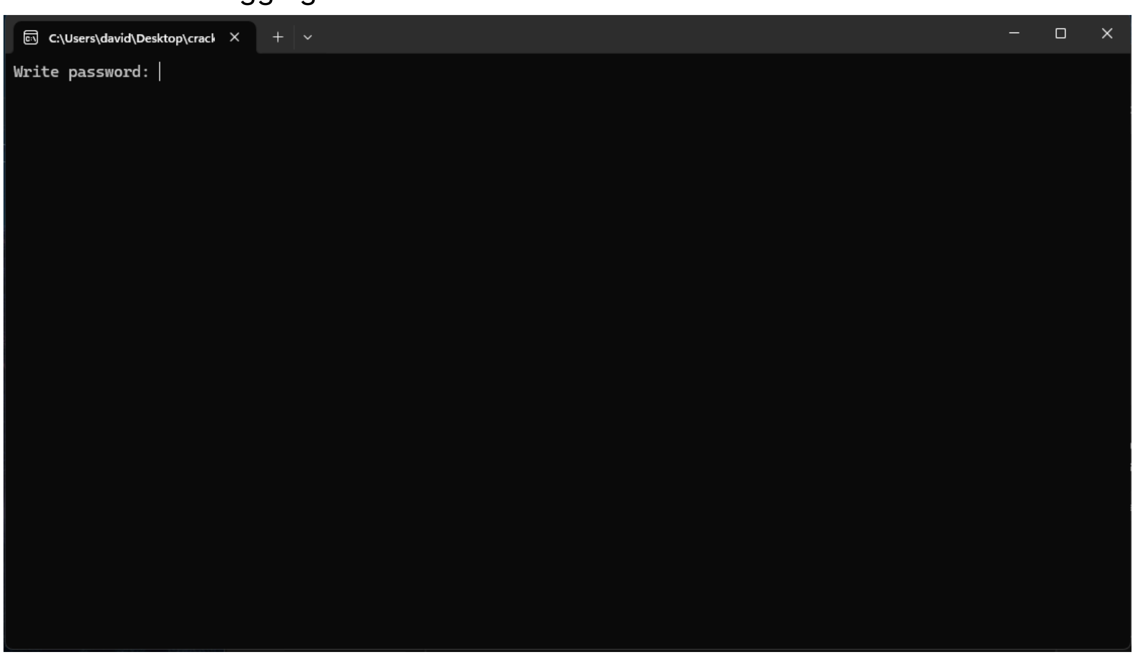
Here I can see the call to IsDebuggerPresent and then a conditional jump that checks if a debugger is present. If one is, it will skip the user input code path.



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.*



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

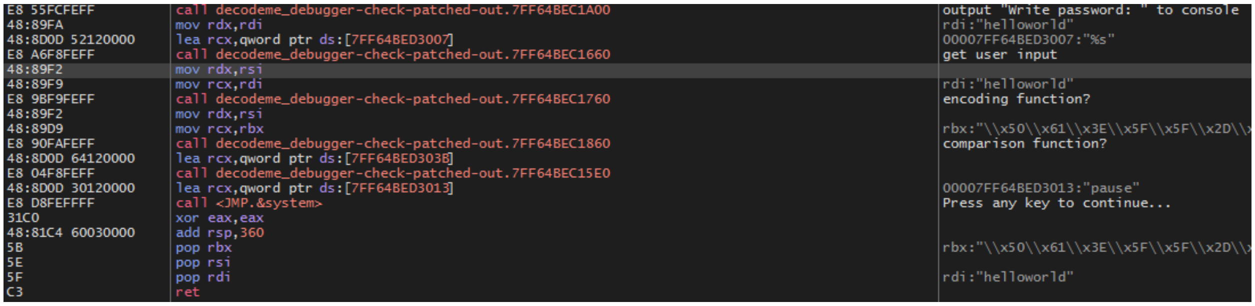


Now just to be extra cautious, I am going to add the breakpoint for kernel32.IsDebuggerPresent back in just to see if any other calls are made to it. No other hits, very nice! Now it is time that I can start looking, probing, and poking around.

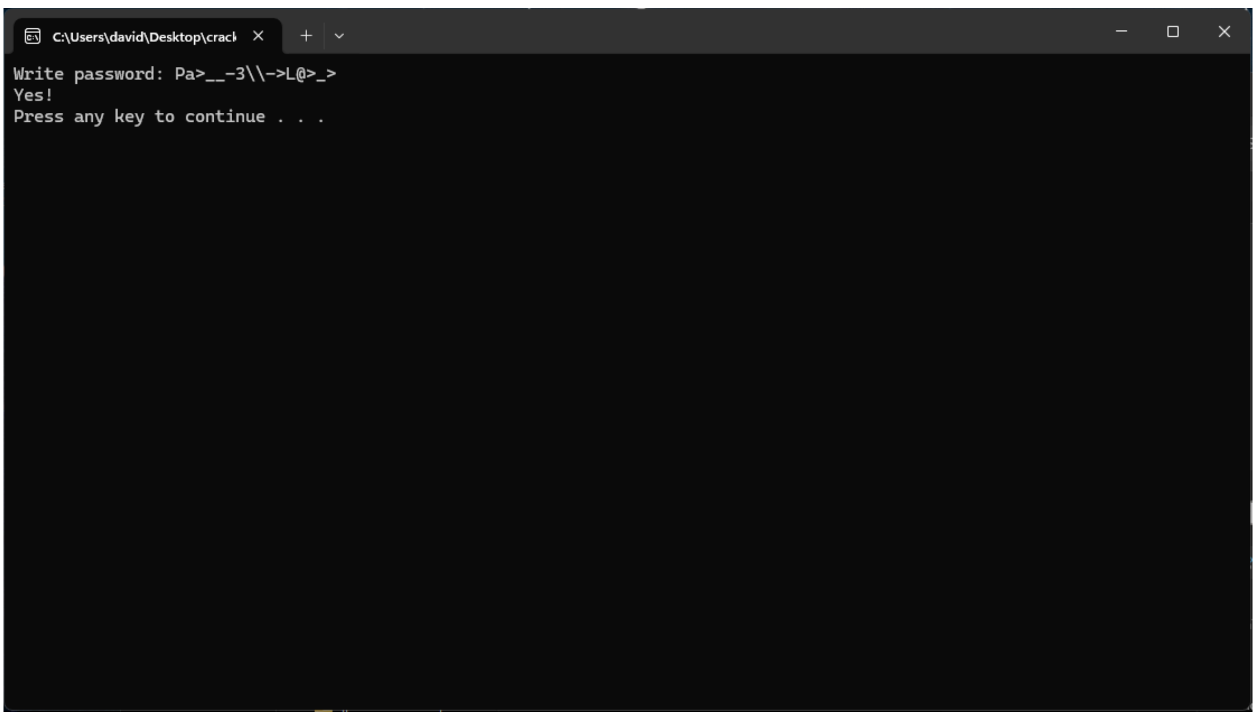
After some stepping through some I land on what appears to be the main function.

```
push rdi
push rsi
push rbx
sub rsp,360                                             main function?
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]                     00007FF64BED302A:"Pa100-322-1L@101"
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   output "Write password: " to console
mov rdx,rdi
lea rcx,qword ptr ds:[7FF64BED3007]                     00007FF64BED3007:"%s"
call decodeme_debugger-check-patched-out.7FF64BEC1660   get user input
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:[7FF64BED303B]
call decodeme_debugger-check-patched-out.7FF64BEC15E0
lea rcx,qword ptr ds:[7FF64BED3013]                     00007FF64BED3013:"pause"
call <JMP.&system>
xor eax,eax
add rsp,360
pop rbx
pop rsi
pop rdi
ret
```

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.



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

__Supposed encoding 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             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) 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 all this function is indeed the encoding function.

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	&#127;	DEL

Source: www.LookupTables.com

After the encoding step, 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!”.

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 rdx,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.