### **Shayan Ahmed Khan**

all all and

NER' REPORT

CIRC CORR

EVERE EVERALING

San Strand

2441CANTIN 100

REEIVERIG

Te traises statements to crobert at

FRESON CHURCHER FRESHILLS

Sr. Threat Researcher

[LinkedIn] [Medium] [Github] [Website]

ave.

N. O. Sanahar

# **Cracked Haven! Why you should avoid using cracked software and how to check their integrity?**

In this age of digital world, everything depends on software. From communication to management to business and operation, software is now an integral part of the modern world. While some of these software are free and open-source while others are sold as a product to the customers and whenever there is money involved in any business, a form of piracy emerges. Software piracy has become very common. Let's be honest we all have used a pirated software in our lives more than once. While there is a huge joy in getting a free cracked version of an invaluable product, there are some risks associated with it. A cracked software might contain harmful arbitrary code in it. Since, it is users own choice to install a cracked software therefore hacking that user is very very easy. In this blog we will look at how software is being cracked, we will check the integrity of cracked software to see whether they contain anything malicious or harmful in it and finally we will try to add our own arbitrary code inside the cracked products.

I will demonstrate the example of a cracked **Adobe Photoshop CS6**, that I found from a pirated website. I will show how this software has been cracked, and I will check the integrity of this cracked software that will ensure if anything malicious had been added in the crack or not. For clearly understanding this blog, you need to have a background knowledge of reverse engineering, x86–64 assembly language, Windows APIs, disassembler tools like IDA pro, and basic knowledge of binary patches and binary patch diffing.

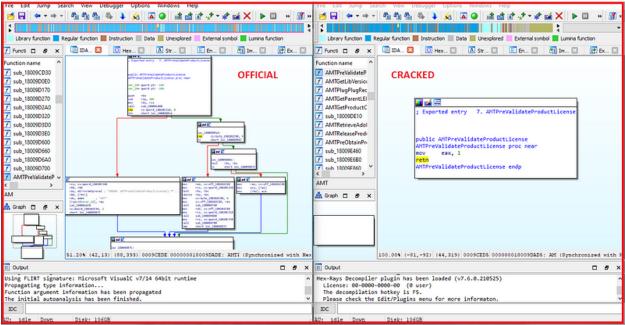
## Disclaimer: Before moving on to the blog. I want to clarify one thing. This is only for educational purposes. I do not support piracy in any case. I do not want my blog to be used for illegitimate purposes. Everything that you do, it is your sole responsibility.

In my foolish younger years before coming to the cybersecurity I've been using a cracked Photoshop for many years. I decided to check its integrity and to found out if I was hacked in any case using that crack. Since, I have been working on finding zero-days and 1-day vulnerabilities using the methodology of <u>root cause analysis</u>, therefore the first thing that came to my mind is to check the difference between the official version and the cracked version. I'm using <u>IDA pro</u> as my main disassembler and an added plugin called <u>BinDiff</u> for binary diffing. IDA pro is a commercial software and it is very expensive, but there is also a free version available which is called IDA Freeware. In this exercise we are using the free version only.

Ubrary function E Regular f		0.224		ed Fun		External symbol 🔜 Lu Primary Unma	🕅 Sta 🖸 🛛 🦹 Secondary Unma 🖸	🔯 IDA V 🖸 🚺 Hex	🕄 🗚 Stru 🖸 👬 En 🕻
unction name		^	Similarity	Confid (	Change		Name Primary	EA Secondary	Name Secondary
sub_180001000		100	1.00	0.14		0000000180125700	sub_100126700 sub_180145060	000000180125700	sub_000000180126700 sub_0000000180145060
sub_180001080			1.00			0000000180145090	sub_180145060 sub_180145090	0000000180145090	sub_0000000180145090
sub_180001290			1.00			0000000180145690	sub_180145690 sub_180146600	0000000180146600	sub_000000180145690
sub_180001330			1.00	1222		0000000180146780	sub_180146780	000000180146780	sub_00000001801467B0
sub_1800013F0			1.00	1222		0000000180146D80	sub_180146D80	0000000180146D80	sub_0000000180146D80
sub_180001410			1.00	1000		0000000180146E50	sub_180146E50	0000000180146E50	sub_0000000180146E50
sub_180001430			1.00			0000000180146EA0	sub_180146EA0	0000000180146EA0	sub_000000180146EA0
sub_1800014A0			1.00			0000000180146EF0	sub 180146EF0	0000000180146EF0	sub_0000000180146EF0
sub_1800014C0			1.00			0000000180146F30	sub_180146F30	0000000180146F30	sub_000000180146F30
sub_1800014F0			1.00			00000001800DC0C0	sub 1800DC0C0	00000001800DC0C0	sub_0000001800DC0C0
j_strdup			1.00			0000000180108AB0	sub_180108AB0	0000000180108AB0	sub_0000000180108AB0
sub_1800016D0			1.00	100.000		00000001800DC4F0	sub 1800DC4F0	00000001800DC4F0	sub_0000001800DC4F0
sub_180001860		12	1.00	1000		0000000180136620	sub 180136620	0000000180136620	sub_000000180136620
		>	1.00			00000001800DBF70	sub 1800DBF70	00000001800DBF70	sub_0000001800DBF70
ine 5453 of 5453			1.00			00000001800DC1E0	sub 1800DC1E0	00000001800DC1E0	sub_00000001800DC1E0
			1.00			00000001800DC6E0	sub 1800DC6E0	00000001800DC6E0	sub_0000001800DC6E0
Graph overview	08	×	1.00			00000001800DC370	sub 1800DC370	00000001800DC370	sub_00000001800DC370
\$			1.00			00000001800D57A0	sub 1800D57A0	00000001800D57A0	sub_00000001800D57A0
夏日	1		0.28		51E	000000018009DAD0	AMTPreValidateProductLicense	000000018009DAD0	AMTPreValidateProductLicense
E			0.06		SIE	0000000180088520	sub 180088520	0000000180088520	sub 0000000180088520
		_							-
			Line 5034	of 5066					
Output									

Binary Patch Diffing of benign and cracked amtlib.dll

In photoshop CS6 amtlib.dll is the library that is used for validating licenses and for cracking photoshop, you just need to replace this library with the cracked library. But how to be sure that there is no harmful code in the cracked version? Well, using the technique of binary patch diffing by the plugin **BinDiff** by zynamics, I've found that only two functions were changed that are shown in the screenshot above. So, by comparing the changes made in these two functions we can find out how the software was cracked. A side-by-side comparison of both functions is listed in the screenshot below:



Comparison of changed function 1 of the cracked binary

As you can see in the official version the CFG shows a series of different code blocks executing after checking different conditions but on the other side in the cracked binary, there is a simple piece of code that is always returning 1 and no licenses are being validated. Let's look at the other function that was changed in the cracked binary.

File Edit Jump Search View Debugger Options Windows Help	File Edit Jump Search View Debugger Options Windows Help
🖆 🖬 🍬 - キー 🎭 🎭 🌭 🔰 🎪 🛋 🛛 🎿 🏦 🖉 - 🖈 🐋 🗙 🕨 💷 . » 🗊 »	🖆 🖬 🗇 - 🔿 - 🆓 🆓 🆓 🦜 🔰 🎪 🖾 🖓 📾 📾 🛣 🖑 🗸 🕨 💷 🔹 🗊 -
Library function Regular function Instruction Data Unexplored External symbol Lumina function	Library function Regular function Instruction Data Unexplored External symbol Lumina function
7 Functi 🗆 🧬 🗴 🕕 10A 🚨 🔘 Hex 🕄 🛋 Str 🗵 👫 En 🕄 📆 Im 💟 📝 Ex 🔇	7 Functi 🗖 🧬 🗴 🕕 10A 🚨 🔘 Hex 🕄 🔺 Str 🗵 🛄 En 🗵 📢 Im 🖾 💓 Ex 🗵
I sub_100030100       Var_38= qword ptr -38h         I sub_10003700       Var_38= qword ptr 18h         I sub_10003700       Path rti         I sub_10003310       Path rti         I sub_10003520       Path rti <td< td=""><td>Function name Judy 18008610 Judy 18008760 Judy 18008760 Judy 18008760 Judy 18008700 Judy 180088100 Judy 18008810 Judy 18008850 Judy 1800885</td></td<>	Function name Judy 18008610 Judy 18008760 Judy 18008760 Judy 18008760 Judy 18008700 Judy 180088100 Judy 18008810 Judy 18008850 Judy 1800885
in         in<	100.004 (-184,-73) (45,236) 00087520 000000180088520: st (Synchronized with 1
🖬 Output 🗖 🗗 🖉 🗙	🖬 Output 🗖 🗗 🛪
Using FLIRT signature: Nicrosoft VisualC v7/14 64bit runtime ^ Propagating type information Function argument information has been propagated The initial autoanalysis has been finished.	Hex-Rays Decompiler plugin has been loaded (v7.6.0.210525) License: 00-0000-0000-00 (0 user) The decompilation hotkey is F5. Please check the Edit/Plugins menu for more informaton.
IDC	IDC
10: idle Down Disk: 10668	AU: idle Down Disk: 10668

Comparison of changed function 2 of the cracked binary

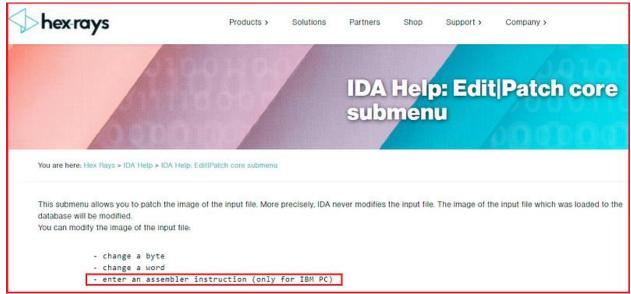
Similarly in the second function, the cracked binary is always returning 0 without checking any conditions. Looks like this function is a pre-condition before validating the licenses. So for the

other function to execute, this function must return 0, therefore in the cracked version it is always returning 0. Luckily this crack doesn't have any harmful code added in it which I have been using in the past. Using a cracked software is never a good idea, because it has been tampered with and attacker can add anything in it. We can check the integrity of cracked software by using the technique of binary patch diffing as demonstrated in the example above.

Now I will demonstrate why **using a cracked software is dangerous** and how easy a user can be hacked with the help of cracked software. We know what functions are used for the crack; we now understand how the crack works. Follow me through this blog to see why a cracked software is harmful for you. I will take the official benign binary and crack it but also add some harmful code in it. The functions that I need to crack are:

- 1. AMTPreValidateProductLicense
- 2. Sub\_180088520

I will first crack the second function sub\_180088520. It is very simple; I just need to return 0 in this function. For that I will use IDA freeware's built-in assembler. However, IDA is not such a good assembler or patcher. There are better tools for patching binaries then IDA, but I am used to working with IDA pro and freeware. The assembler of IDA is limited to the IBM pc only but we can apply patches in bytes or word.



IDA pro assembler instruction limitation

For sub\_180088520, cracking is very simple, I just needed to assemble instructions using the builtin IDA freeware assembler so that the return value is set to 0. As we all know that the return register is always rax or its counterpart eax for 32-bit value. So, I assembled the value of eax to xor with itself. Anything xor with itself results in the value of 0. And then return instruction will give control back to where this function was called from with the return value 0. See the screenshot below to understand:

5 🗅	Сору	Ctrl+C	r Options Windows Help 🧏 🧟 🖬 🚳 📩 🗟 🖈 🛪 📾	🗙 🕨 🔲 Local Windows debugger 👻 🍖 🛃 🚮 🐩 🎬	
	Begin selection Select all	Alt+L	ction 📄 Data 📕 Unexplored 💭 External symbol	Lumina function	
	Select identifier Undo Redo Export data Code Data Struct var Strings Array Undefine Rename	Shift-Enter Ctrl-Z Ctrl-Y Shift-E C D Alt+Q Numpad+* U N	IDA Vien-A         Image: Comparison of the comparis	align 28h SUBROUTINE b_180088520 proc near ar_36 = quord ptr Address : 0x1 : 0x10008520 revious lne: Address : 0x1 : 0x10008520 revious lne: Address : 0x1 : 0x10008520 revious lne: Concert Concert Conce	
ध स स स स स स	Operand type Comments Segments Structs Functions	• • •	.text:000000130085520 .text:00000013008520 .text:000000130085523 .text:000000130085523 .text:000000130085523 .text:000000130085528 .text:000000130085520	push         text:0000001500E520         push           push         rbx         text:0000001500E520         xor           push         rsi         text:0000001500E520         xor           push         rdi         text:0000001500E520         xor           sub         rsp, 40h         text:0000001500E523         retn           mov         rdi, cdx         text:0000001500E523         retn           test         cdx, rdx         test         cdx, rdx	
u u	Patch program Other	,	Change byte 1 Change word	jnz short loc_18083853A lea rsi,aObtain ; "Obtain" jmp short loc_18008854F	
ne 1410 c Output	Plugins f 5453		Assemble Patched bytes Ctrl+Alt+P Apply patches to input file	: sub_100008520 (Synchronized with Hex View-1)	8
opagat	IRT signature: A ing type informa argument inform ial autoanalysis	tion ation has been			

Cracking 2nd function in IDA pro

Now cracking the first function is also very easy, we just need to return always 1 value. Same approach can be applied to it, instead of xoring we just need to move an immediate value 1 in the eax register and return. But I'm demonstrating how a **hacker** and **software cracker** can add harmful code in a cracked binary. So, I decided to do something extra with the first function. I looked at all the imports that this binary is using and I saw a message box import as well. **I decided to show a message box whenever this cracked binary is used**. For using an API import there are some requirements:

- 1. Must know the address of API import
- 2. Must calculate the offset of calling that API import
- 3. Must push all the parameters onto the stack frame
- 4. Avoid corrupting stack frames by equal pushing and poping
- 5. Must calculate offset for the string values used for parameters
- 6. Calculate opcodes for machine instructions because we must change bytes

For calling any instruction, we need its opcode because we cannot directly assemble an instruction in IDA freeware or pro because of its assembler limitations. There is a way to calculate opcodes for every instruction, but I prefer doing smart way. I write the code in **visual studio** and **disassemble** it to know its opcodes or for complex code I also write the code in assembly and using MASM assembler I debug and find its opcode. This is a very simple example of just calling a message box therefore I will write code in basic c instead of assembly language.

Γ	int (	t main	()			
	ι 40	55			push	rbp
	57				push	rdi
	48	81 EC	E8	00 00	00 sub	rsp,0E8h
L	48	8D 6C	24	20	lea	rbp,[rsp+20h]
	48	8D ØD	5C	D6 00	00 lea	rcx,[BB32171F_Source@cpp (07FF74D411012h)]
L	<b>E8</b>	88 D9	FF	FF	call	<pre>CheckForDebuggerJustMyCode (07FF74D401343h)</pre>
		//Wi	nExe	c("cmo	l.exe /c ed	ho YOU HAVE BEEN HACKED > HACKED.txt", 1);
		Mess	ageB	oxW(0,	, TEXT("BEW	WARE"), TEXT("You Have Been Hacked!!!"), 0);
Þ	45	33 C9			xor	r9d,r9d
	4C	8D 05	FB	64 00	00 lea	r8,[string L"You Have Been H\x4000\0\0\0\0" (07FF74D409EC0h)]
	48	8D 15	E4	61 00	00 lea	rdx,[string "BEWARE" (07FF74D409BB0h)]
L	33	C9			xor	ecx,ecx
L	FF	15 7C	C7	00 00	call	<pre>qword ptr [imp_MessageBoxW (07FF74D410150h)]</pre>
L		//Ex	itPr	ocess	(0);	
	}					
	33	C0			xor	eax,eax
		8D A5	C8	00 00	00 lea	rsp,[rbp+0C8h]
	5F				рор	rdi

Opcodes generated by visual studio debugger

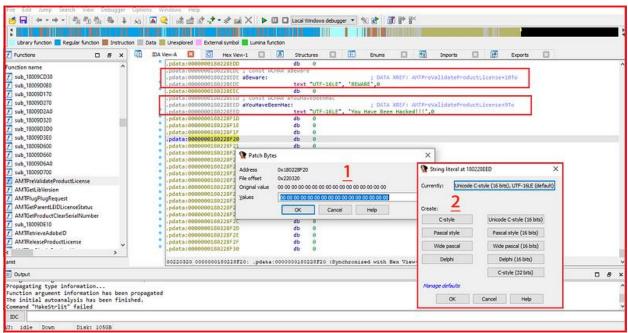
In the screenshot above, all the opcodes needed for calling message box API are generated by the visual studio debugger. We can get the idea of how to change bytes using this approach. I will patch the first instruction that is *xor r9d, r9d* and the opcode for this instruction is *45 33 C9*. This is the first parameter for message box API and I'm passing it a value of 0 by xoring r9d register.

	11111									1 A A	
Library function 📃 Regular function	n 📕 Instru	ction 📰 Data 🔳 Unexp	iored 📕 Exter	nal symbol 📕 Lumin	a function	n					
7 Functions		IDA Ven-A		Hex View-1	A	Structures 🖸	E	00000010000CA	A CC 23 54 20 68 66 48 51 C4 46 62 68 69 38 52 50 58 C3 CC CC CC CC CC CC CC CC CC	41 90 97 1et	AND I SHOW
Function name		000000120000A90	FF EB 03	83 C8 FF 48 88	80 24	38 82 88 88 48 33 82 88 88 49 41 50 5F	Se.fEyH	0000001580304	C4 E9 48 FL FF FF CC CC CC CC CC CC CC CC	FF 45 83 #Ser/1013/46+	-operation of the second se
Junknown_libname_1275		000000010000000000000000000000000000000	SE 50 58	C3 CC CC CC CC	CC CC		^)[A111	00000010000CA	20 30 32 18 16 00 00 75 72 48 00 51 18 70 33 09 48 80 05 97 00 10 00 45 17 41 01	48 00 55 36x	L.M.
7 sub_18009C1F0		00000018009DAC0	E9 48 FE	FF FF CC CC CC	CC CC	CC CC CC CC CC CC	ékbyy11	00000013000000 00000013000000	00 4C 9C 87 00 48 89 44 24 20 18 02 CF 01 10 TC 51 16 00 02 00 00 00 18 62 80 30 6F	00 C7 05 La., ML05 -001 51 16 00 10	
7 sub_18009C290		000000018009DAD0	40 53 48	83 EC 36 48 88	09 68	A2 30 F6 FF 48 83	@SHf10H	0000001000000	20 00 74 49 48 88 05 15 51 15 00 48 88 00	67 68 16 . 1044 0. MA	Chu I
7 sub 18009C2C0		000000018009DAE0 000000018009DAE0	30 32 E8 33 C9 48	16 00 00 75 32	48	Data format	·	446664336669CB	10 09 05 42 51 26 00 28 55 94 02 00 48 55	25 36 55 k.60eut	90,0
7 sub_18009C3A0		000000150090470	4C 9C 0F	00 48 89 44 24	28	Columns	* HED	04000110001500005	15 00 03 17 16 00 18 15 16 00 18 72 10 10 10 00 03 17 16 00 18 46 94 81 00 18 18 46	17 45 58	N.Ot
7 sub_18009C8C0		0000000180090810	7C 51 16		68	Text		0000001000000	70 00 48 80 05 10 51 16 00 80 00 80 00 18 00 77 48 58 00 88 57 16 00 48 80 05 45 00	er 10 F6 .Hr. Q. F.B. 18 99 67 SHL C. H. I	
J sub_18009CAE0		0000000180090820		48 88 05 58 51	15			0000001800109	100 14 00 4C 4D 4D 27 84 11 40 46 40 15 80	ES 01 00 .AL	
1 sub_18009CD30		00000010000030	00 48 85		C6	and the second se	F2 1	0000001300000	D0 89 44 24 20 E8 27 CF 01 00 48 83 C4 50	58 C3 CC 805-311	
1 sub_18009D0E0		0000000130090840	89 85 42 16 88 48	51 16 00 E8 55 EB 00 BE 57 16	94	Save to Edit	Q8	0000001000000	10 - 74 42 89 42 40 85 (8 74 83 41 89 66 40 10 11 (8	45 CO 74 T. E.M. AT. 42.	n.ft
F sub_180090170		000000100000000	00 03 E7	16 00 E8 46 94	01		1.87	0000001500903 0000001500903	text:00000018009DAD0	push	rbx
f sub_180090270		000000180000870	08 48 88	05 10 51 16 00	88	Synchronize with	×Q.	0000001000000	text:000000018009DAD2 text:000000018009DAD6	sub	rsp, 30h
		000000180090880	FF 48 88	00 B8 E7 16 00	48	100	1.6	0000001500900	text:000000018009DAD9	call	sub 180001880
7 sub_1800902A0		0000000150090890	14 C0 4C	80 00 27 8A 11	99	Font	1.1.5	00000010000CC	text:00000018009DADE 5	c=p	cs:qword_18020C318, 0
7 sub_18009D320		0000000180090880	89 44 24	20 F8 27 CF 01	00 42	83 C4 30 58 C3 CC	205 è 1.	HEADLAT	text:00000018009DAE6	jnz	short loc_18009D81A
7 sub_18009D3D0		00000001800008C0	48 85 69	74 86 67 81 86	00 00	08 33 C0 48 85 D2	H_Ét.C.		text:000000018009DAE8 text:000000018009DAEF	mov xor	rcx, cs:qword_18020C34 r9d, r9d
1800903E0		000000180000000	74 02 89	02 40 85 C0 74	83 41	89 88 4D 85 C9 74	t.k.M.At		text:000000010009DAF2	lea	rax, afrrorAmtpreval ;
J sub_180090600		000000130090850	BA 48 88	05 80 50 16 00	49 89	01 F3 C3 CC CC CC		It.6Alll	text:00000018009DAF9	lea	r8d, [r9+1]
7 sub_180090660		00000001000000000	48 89 5C 7C 3C F6	24 08 57 48 83 FF 48 88 00 00	EC 30 E7 16	88 DA 48 88 F9 E8 88 48 85 C9 75 2D		filexúmiùè .cH_Éu-	text:00000018009DAFD	lea	rdx, aAmt ; "AMI
J sub_18009D6A0		0000000100090C00	44 80 41			00 48 8D 05 46 68	D.A.HC.S		text:00000018009DB04	mov	[rsp+38h+var_18], rax
7 sub_180090700		000000018009DC20	10 00 48		88 45	33 C9 48 89 44 24		.E3ÉHIDS	text:000000018009D809 text:000000018009D80E	call	sub_18008AAE0 cs:dword_180202C94, 2
AMTPreValidateProductLicense		000000018009DC30		CE 01 00 88 FF	69 69	00 EB 00 44 58 C3	-ètî	A.0.3.	text:000000100090000	imp	short loc 18009087C
California	>	0000000180090C40	48 88 D7	E8 88 93 FE FF	88 08	E8 41 3C F6 FF 48	нахёттру	o deakayn		24	
AM		0009CED8 0000000	18005DAD8:	AMTFreValidate	Product	License+8 (Synchros	nized with	IDA View	(~A)		
Dutput											
Using FLIRT signature: Micro Propagating type information Function argument informatic The initial autoanalysis has IDC	n has bee	n propagated	ntime								

Patching instructions using opcodes in Hex-View

One thing of IDA pro or freeware is very good that it gives us live mapping of every instruction in the Hex-format as well. Just click on the instruction you want to see in hex and change tab from IDA view to Hex-view. I edited the bytes that were mapped on the instruction that I wanted to change and saved the IDA image. In the 3rd section of this screenshot, you can see the instruction

has been changed to whatever we wanted. I will use same approach for patching other instructions as well. For the second instruction I needed to create a string variable in the data section that must be called in a register as second parameter. So, I found an empty buffer space in data section and edited those empty bytes with the bytes that I wanted as my string. In the screenshot below you will understand how to create variables.



Creating variables for parameters to be used for Message Box

I created two variables at an empty buffer in the data segment of this binary. One of the variables is used for text shown inside a message box and other variable is used for caption of the message box. Creating these variables is very easy, just find an empty buffer address and change bytes from the edit menu. As shown in the screenshot above, you can patch bytes and save the data that you want to save which in this case is the hex of strings that I want to save. You must also keep in mind the variable types being used here. Message box API uses wide strings therefore I've converted my variable to the Unicode 16-bit style instead of Ascii that are 8-bits. The hotkey for defining string literals is ALT + A.

I've defined the string variables, now the remaining task is to load these offset values in registers used for passing parameters. For that I will provide a simple formula:

#### Offset = (callee\_address - caller\_address - instruction size)

My string text is on the address **180228EED** which is callee\_address. The address from which I want to call and save it in register is on the address **18009DAD9** and the instruction size is **7**. Put the values in the formula and we get the offset address that is **18B40D**. From the opcodes generated by the Visual studio, I know the opcode for lea starts with 48 8D, 15 is the register in which the value must be moved and rest is the offset address in little Endian format. So, the complete opcode for the instruction: *lea rdx, offset:text* is **48 8D 15 0D B4 18 00**. I will save both strings in registers

by calculating their offset values and saving in registers using the LEA instruction as shown in the screenshots below:

.text:000000018009DAD0	; ====================================
.text:000000018009DAD0	
.text:000000018009DAD0	
.text:000000018009DAD0	public AMTPreValidateProductLicense
.text:000000018009DAD0	AMTPreValidateProductLicense proc near ; DATA XREF: .rdata:off_1802014C8↓o
.text:000000018009DAD0	push rbx
.text:000000018009DAD2	sub rsp, 30h
.text:000000018009DAD6	xor r9d, r9d ; uType
.text:000000018009DAD9	lea rdx, <mark>aYouHaveBeenHac</mark> ; lpText
.text:000000018009DAE0	lea r8, aBeware ; lpCaption

Instructions for loading offset value in registers

000000018009DA90	FF	EB	03	83	C8	FF	48	8B	8C	24	30	02	00	00	48	33	ÿë.fÈÿH∢Œ\$0H3
000000018009DAA0	CC	<b>E8</b>	9A	2D	ØB	00	48	81	C4	40	02	00	00	41	5C	5F	ÌèšH.Ä@A\_
000000018009DAB0	5E	5D	5B	<b>C</b> 3	CC	CC	CC	CC	CC	CC	СС	CC	CC	CC	CC	CC	^][ÃÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
000000018009DAC0	E9	<b>4</b> B	FE	FF	FF	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	éKþÿÿÌÌÌÌÌÌÌÌÌÌÌÌÌ
000000018009DAD0	40	53	48	83	EC	30	45	33	C9	48	8D	15	0D	B4	18	00	@SHfì0E3ÉH´
000000018009DAE0	4C	8D	05	F7	B3	18	00	90	33	<b>C</b> 9	FF	15	F0	8A	ØF	00	L÷³3Éÿ.ðŠ
000000018009DAF0	B8	01	00	00	00	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB00	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB10	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB20	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB30	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB40	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	
000000018009DB50	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	

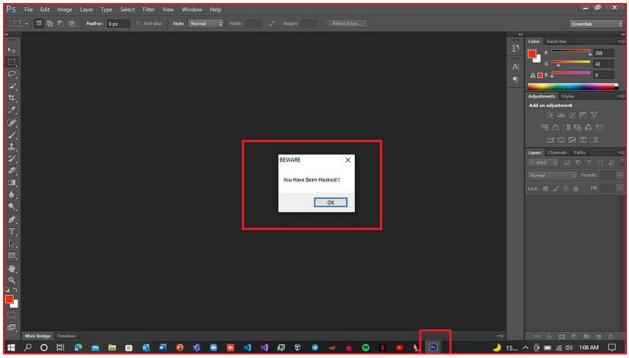
Opcodes for lea from offset value

The last parameter is also 0, so it is easy to assemble. Simply xor the register with itself and 0 will be saved in that register. The last step is to call the message box API. The method for calling it will be same. Need to calculate the offset value and call the API using the opcode for call instruction. The offset that I calculated is 0F8AF0 and the opcode for call instruction is FF 15, put it together and we can call the message box API using the opcode as shown in the screenshot below:

ext:000000018009DAD0	; Exported entry	7. AMTPre	ValidateP	roductLicense							
ext:00000018009DAD0											
ext:000000018009DAD0	; ========== {	UBRO	UTINE						=		
ext:000000018009DAD0	3										
ext:000000018009DAD0	•										
ext:000000018009DAD0	put	lic AMTP	eValidate	ProductLicense	e						
ext:000000018009DAD0	AMTPreValidateProdu	ctLicense	proc near	r ; DATA XREI	F: .rda	ata:off	180201	4C84o			
ext:000000018009DAD	pu:	h rbx									
ext:000000018009DAD2	2 sub	rsp.	30h								
ext:000000018009DAD	xor	r9d	r9d	; uType							
ext:000000018009DAD9	lea	rdx,	aYouHavel	BeenHac ; 1pTe	ext						
<pre>ext:00000018009DAE0</pre>	e lea	r8,	aBeware	; lpCaption	n						
ext:00000018009DAE7	nop										
ext:000000018009DAE8	3 xor	ecx	ecx	; hWnd							
<pre>ext:00000018009DAEA</pre>	a cal	1 cs:/	lessageBoxi	N 2000001200				48 88		02 00 00 48 33	94.fE9H<@\$0
<pre>ext:000000018009DAF0</pre>	mov	eax	. 1	0000001200	MODADO	CC FR 94	20 08 00	48 81	60 24 30	02 00 00 48 55	
ext:000000018009DAF5	5 nop			0000001500	BOADO	SE SD SB	C3 CC CC	CC CC	CC CC CC	CC CC CC CC CC	
ext:00000018009DAF6	5 nop			0000001886	BODACO I	E9 48 FE	FF FF CC	CC CC		CC CC CC CC CC	
ext:000000018009DAF7	nop			3000001500		48 53 48	83 EC 38	45 33	C9 48 8D	15 0D 84 18 00	@SHf10E3ÉH
<pre>ext:00000018009DAF8</pre>	3 nop			0000001800	CONCISION OF CONCESSION	4C 8D 85	F7 83 18	00 90	33 09 00	15 F0 8A 8F 00	L+*3Ey.05
ext:000000018009DAF9				0000001500	90800	98 98 98	98 98 98	90 90	98 98 98	90 90 90 90 90 90	
ext:000000018009DAF/	A nop			0000001800	90810	98 98 98	98 98 98	98 98	98 98 98	98 98 98 98 98	
<pre>ext:000000018009DAFE</pre>	a nop			3000001880	tinese .	98 98 98	98 98 98	98 98	90 90 90	90 90 90 90 90	
ext:000000018009DAF0				0000001300	erenere ;	98 98 98	98 98 98	99 98	98 98 98	90 90 90 90 90 90	
ext:000000018009DAF				0000001800	90858	98 98 98	98 98 98	98 98	90 90 90	90 90 90 90 90	
ext:000000018009DAFE				3000001300	90868	98 98 98	98 98 98	98 98	98 98 98	98 98 98 98 98	
ext:000000018009DAFF	nop			0000001880		90 90 90	90 90 90	90 90	90 90 90	90 90 90 90 90	
				0000001500		98 98 98	98 98 98	98 98	90 90 90	98 98 98 98 98	
	99DAEA: AMTPreValidat	eProduct:	License+1A	(Syr 0000001800		90 90 90 90	98 98 98	98 98	98 98 98	98 98 98 98 98 96	
<				0000001800		98 90 98	98 98 98	98 98			

Instruction patched for calling Windows API imports

Rest of the instructions that were in this function, I simply patched those with NOP instruction which literally means no operation. After calling the Message box API, I've added crack as well which in this case is to return value 1 whenever this function has been called. Then I patched the binary file by applying patched from the edit menu and replaced the official binary with my version of patched binary. Let's see what happens when photoshop is opened.



Cracked photoshop with included code execution

As shown in the screenshot and video above, cracked photoshop pops open a message box whenever the application is launched. The alert says Beware, You have been hacked !!! This is just for proving my point that any kind of code can be executed in the backend when a cracked software is used.

#### Conclusion

Software piracy is very common these days. Even we as software consumers often prefer a cracked software because it's free of cost. However, using a cracked software could be very dangerous. As I have shown in my article, how hackers can easily add malicious code inside a cracked software that will be executed at the back end without user even noticing a thing. Like in this example, what if I called a ShellExecute API or WinExec API or any other API used for creating processes? A simple parameter to these APIs could download and execute another malware on the system whenever this application is opened. You should avoid using cracked software and always go for the premium product from a trusted official vendor.