Shayan Ahmed Khan

Sr. Threat Researcher

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

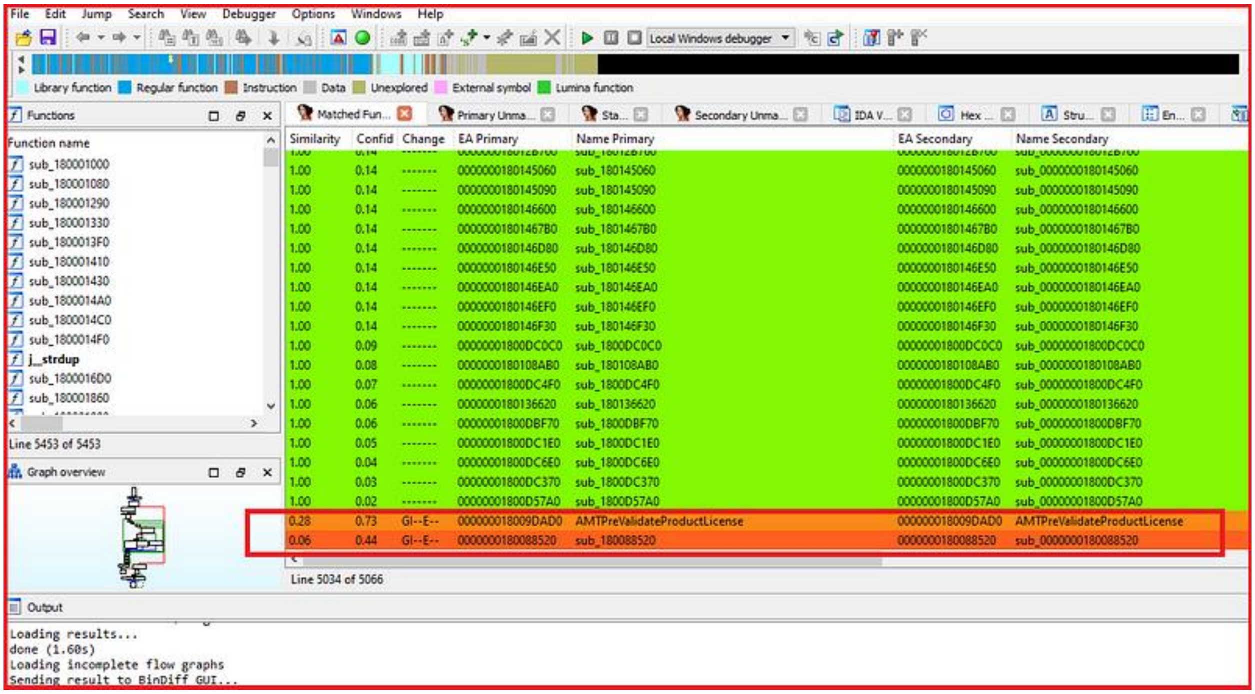
Cracked Haven! Why you should avoid using cracked software and how to check their integrity?

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 software to show how easy it is for software pirates to hack the users who are using their 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.

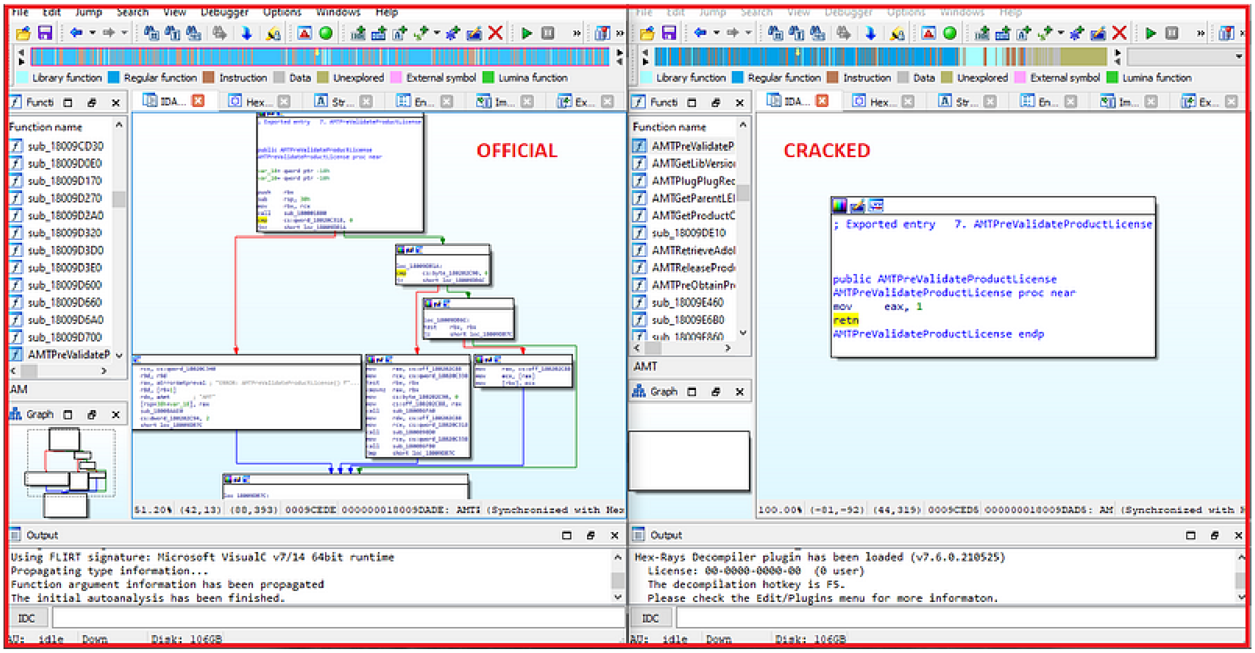
Disclaimer: Before moving on to the blog. I want to clarify one thing. This is only for educational purposes. I do not support piracy in any case. I do not want my blog to be used for illegitimate purposes. Everything that you do, it is your sole responsibility.

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 **root cause analysis**, 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 **IDA pro** as my main disassembler and an added plugin called **BinDiff** 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.



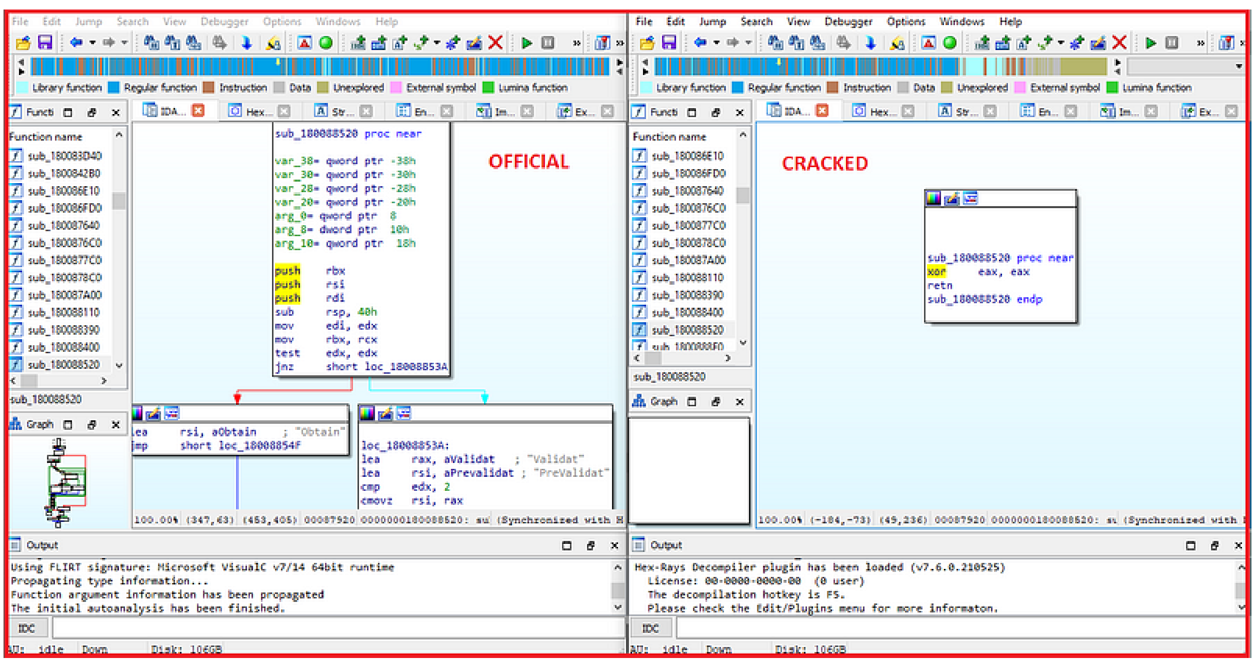
Binary Patch Diffing of benign and cracked amtlib.dll

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.



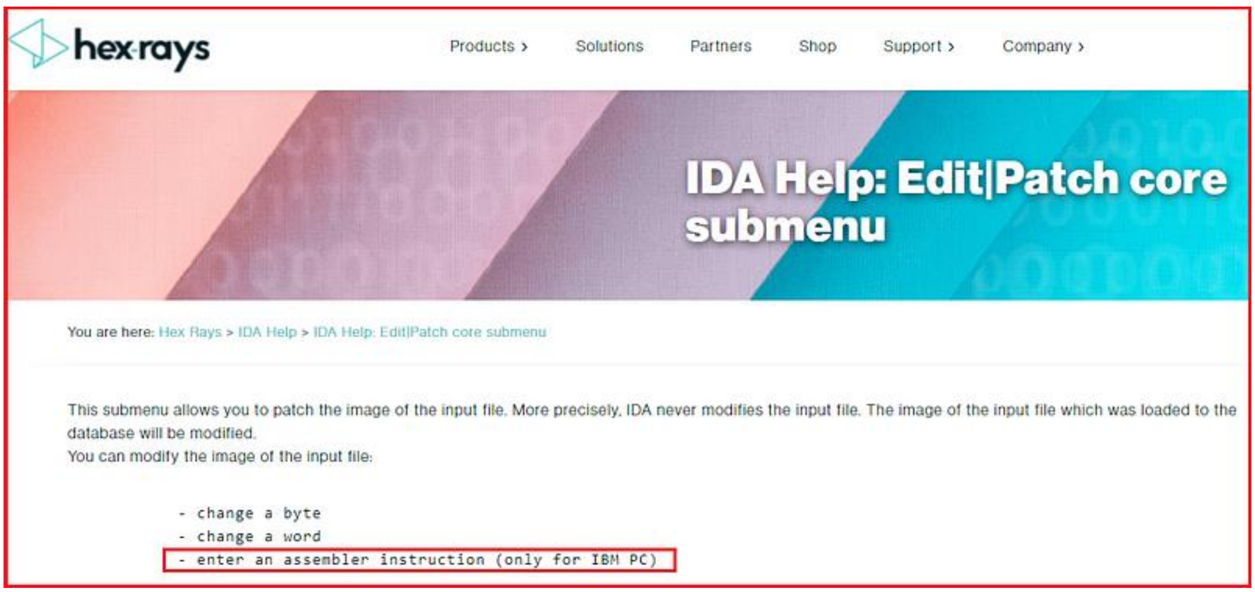
Comparison of changed function 2 of the cracked binary

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:

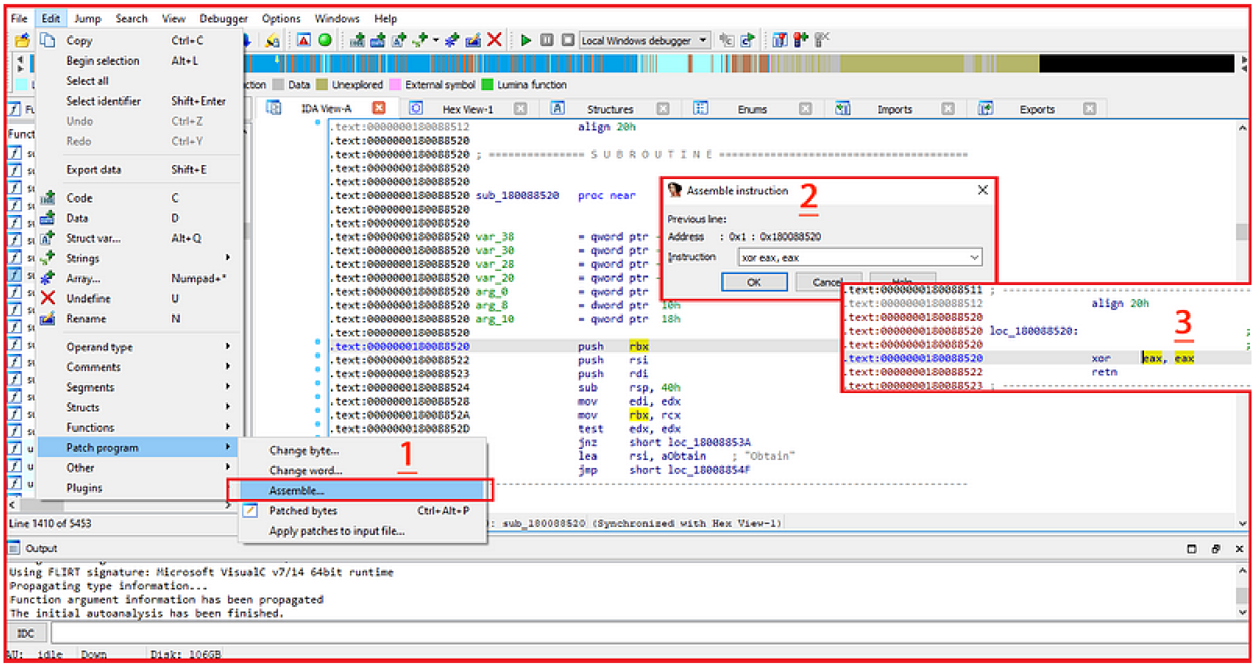
1. AMTPreValidateProductLicense
2. Sub_180088520

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 built-in 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:



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:

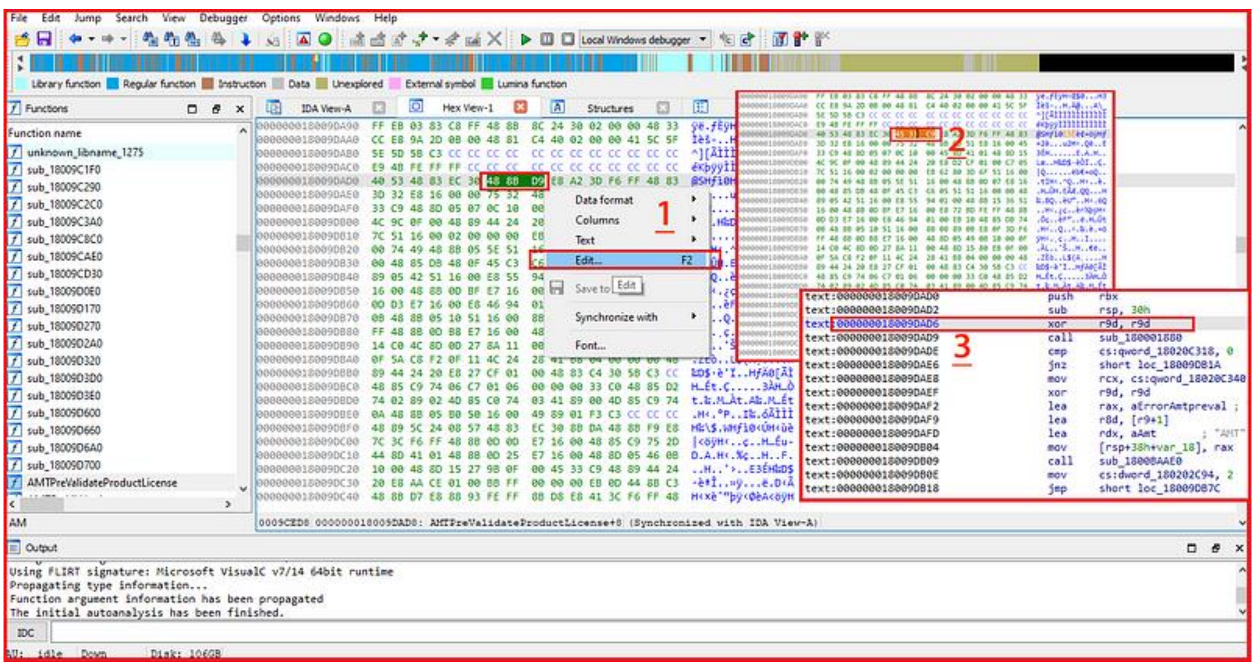
1. Must know the address of API import
2. Must calculate the offset of calling that API import
3. Must push all the parameters onto the stack frame
4. Avoid corrupting stack frames by equal pushing and poping
5. Must calculate offset for the string values used for parameters
6. Calculate opcodes for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

```
int main()
{
40 55                push        rbp
57                   push        rdi
48 81 EC E8 00 00 00 sub         rsp,0E8h
48 8D 6C 24 20       lea         rbp,[rsp+20h]
48 8D 0D 5C D6 00 00 lea         rcx,[__BB32171F_Source@cpp (07FF74D411012h)]
E8 88 D9 FF FF       call        __CheckForDebuggerJustMyCode (07FF74D401343h)
    //WinExec("cmd.exe /c echo YOU HAVE BEEN HACKED > HACKED.txt", 1);
    MessageBoxW(0, TEXT("BEWARE"), TEXT("You Have Been Hacked!!!"), 0);
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)]
33 C9                xor         ecx,ecx
FF 15 7C C7 00 00    call        qword ptr [__imp_MessageBoxW (07FF74D410150h)]
    //ExitProcess(0);
}
33 C0                xor         eax,eax
48 8D A5 C8 00 00 00 lea         rsp,[rbp+0C8h]
5F                   pop         rdi
```

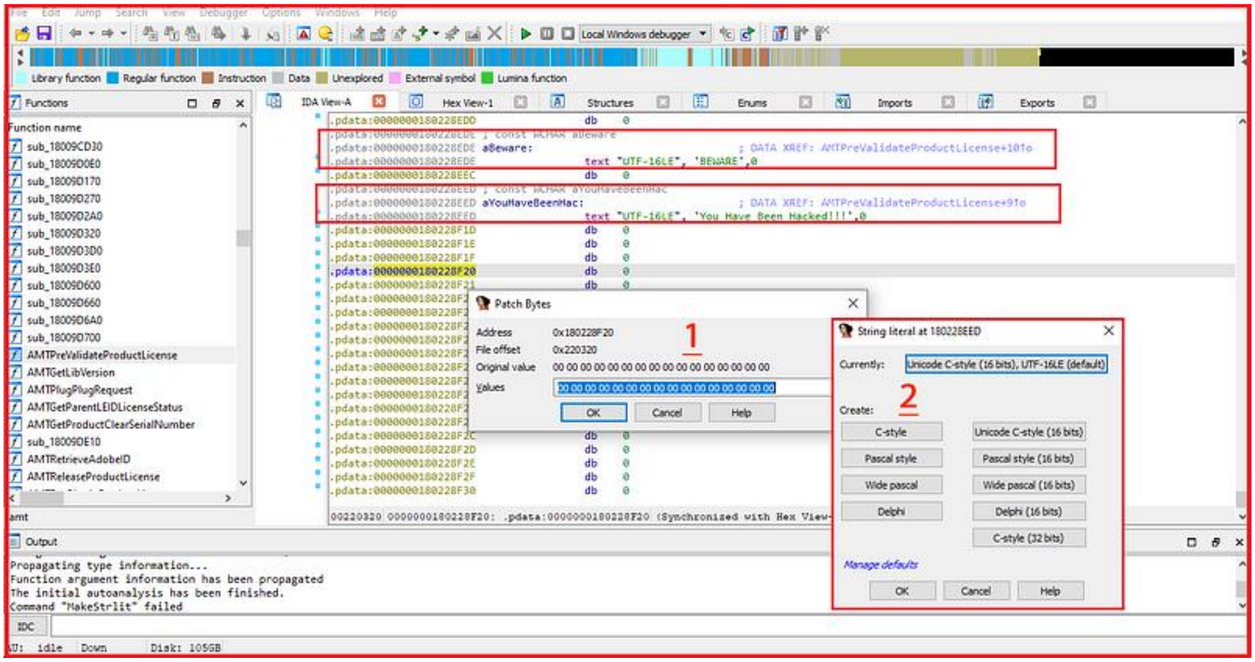
Opcodes generated by visual studio debugger

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.



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.



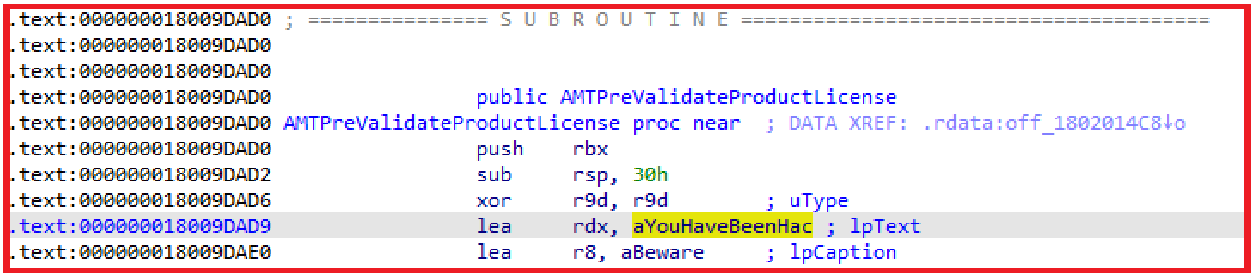
Creating variables for parameters to be used for Message Box

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**.

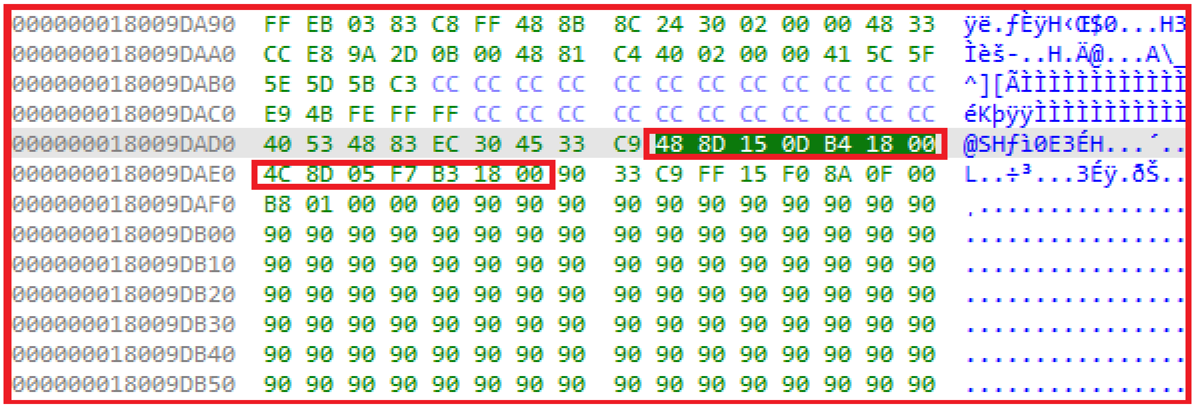
I've defined the string variables, now the remaining task is to load these offset values in registers used for passing parameters. For that I will provide a simple formula:

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:

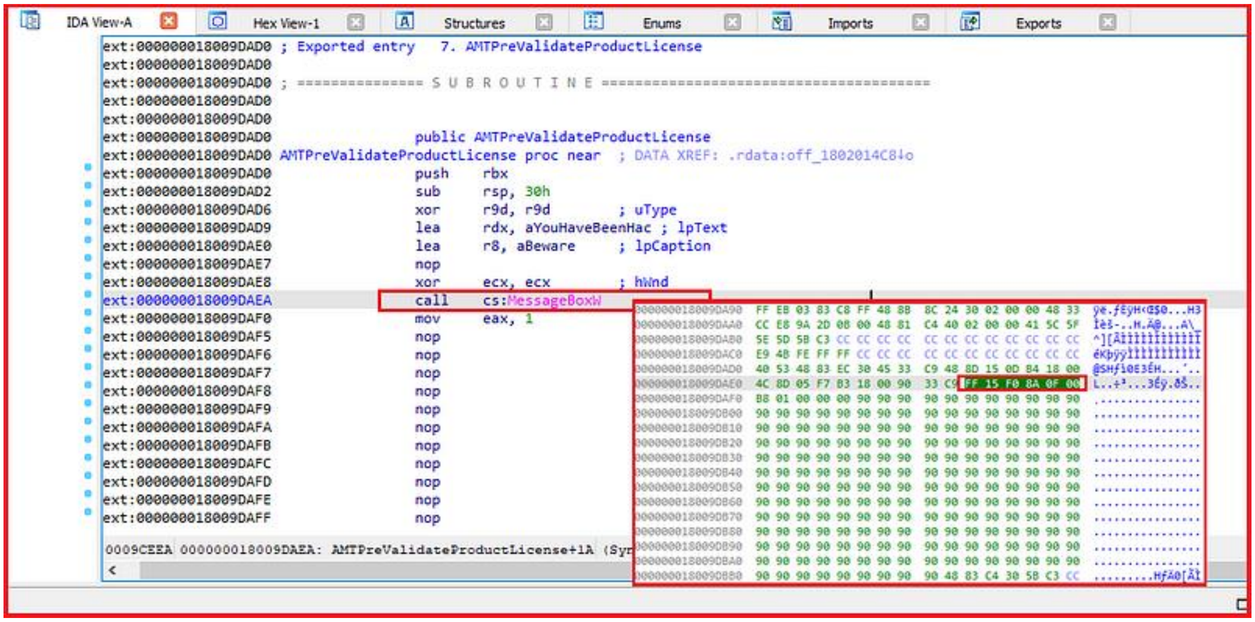


Instructions for loading offset value in registers



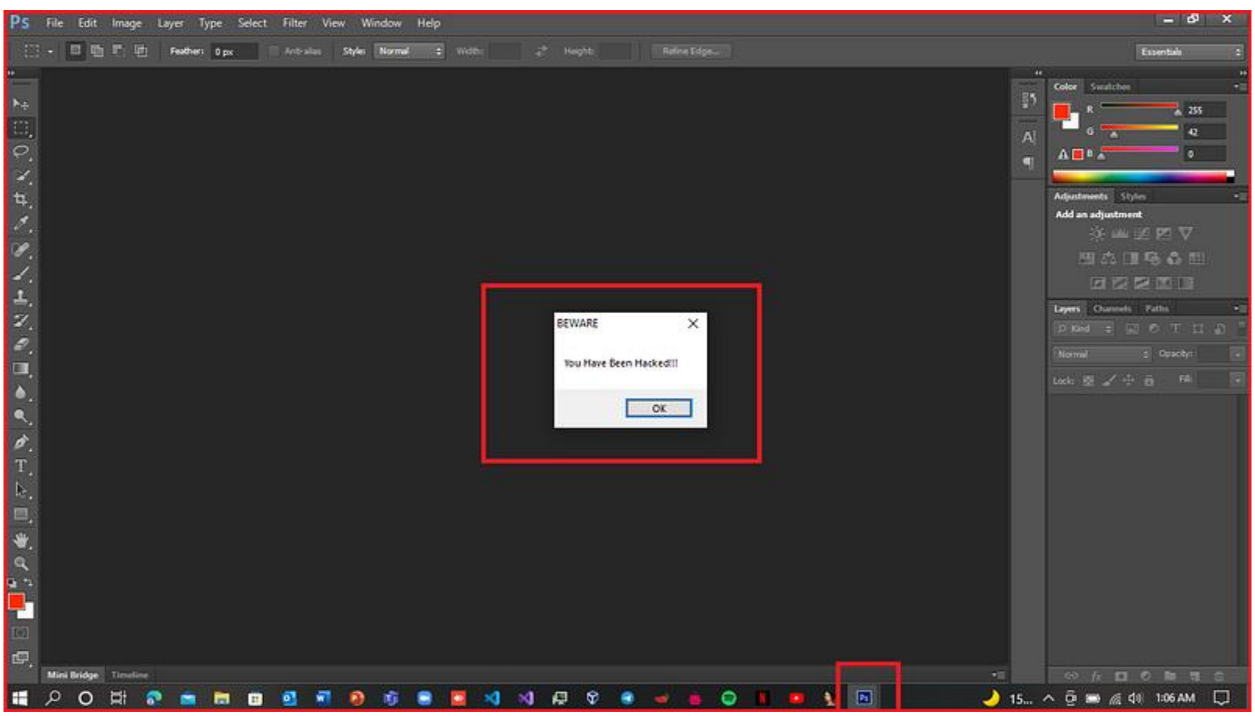
Opcodes for lea from offset value

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:



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.