Assignment 3 Report

Dimitry Rakhlei

Contents

[Summary 3](#_Toc528370744)

[Process 3](#_Toc528370745)

[Analysis 3](#_Toc528370746)

[crackme1.exe 3](#_Toc528370747)

[Unsuccessful Authentication: 4](#_Toc528370748)

[Successful Authentication: 4](#_Toc528370749)

[Patch: 5](#_Toc528370750)

[crackme2.exe 5](#_Toc528370751)

[Unsuccessful Authentication: 6](#_Toc528370752)

[Successful Authentication: 7](#_Toc528370753)

[Patch: 8](#_Toc528370754)

[Conclusion 9](#_Toc528370755)

# Summary

The goal in this assignment was to reverse engineer executable programs and extract key information and disable specified portions of their code. The software used was x96dbg (x64 & x32dbg) which is an open source version of OllyDbg. They both look and perform very similarly.

# Process

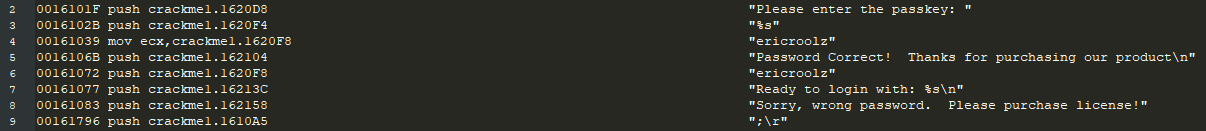
The debugging process was performed inside of a VMware virtual machine. The machine was configured, and a snapshot was created in case of a malware infection.

For each executable I looked at hard coded strings to get a sense of where in the code certain printouts occurred. This was done by right clicking on the disassembled code and selecting Search For -> All Modules -> String References.

Before doing any modifications to the code it is nice to also see what kinds of functions were loaded and used by the process. Here we use the feature found by right clicking on the disassembled code and pressing Search For -> All Modules -> Intermodular Calls. The output tables can be found in the root directory submitted with this report. The log files are called *“function\_calls\_crackme1.txt”* and *“function\_calls\_crackme2.txt”* for the function calls and *“string\_references\_crackme1.txt”* and *“string\_references\_crackme2.txt”* for string references.

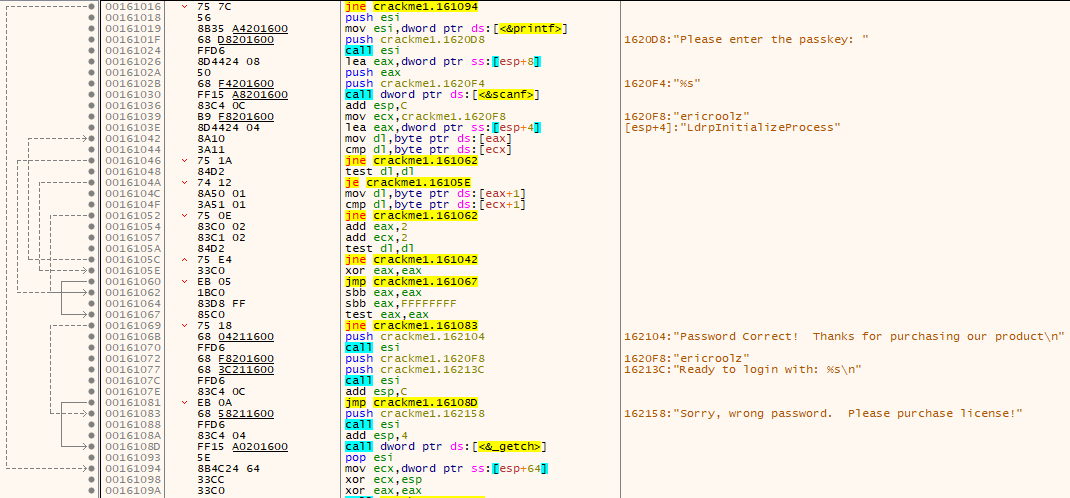
# Analysis

## crackme1.exe

After looking at the function calls of *crackme1* it appears that the application uses scanf and printf functions and does not load any graphical functions. It may very well be a console application which asks for input and has outputs displayed as a response. When we looked at the strings inside of *crackme1* we found that it contains strings such as:

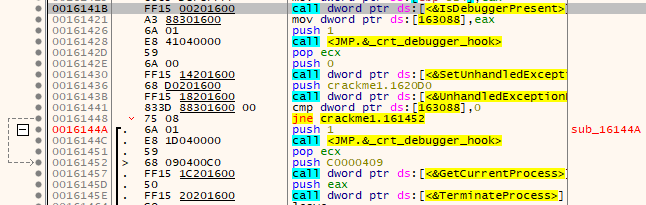
We will follow the string which says, *“Password Correct!”* and see what comparison instruction causes this code to execute.

Here is a portion of the code which will perform the authentication:



At address 00161030 scanf is called with the parameter “%s” meaning that a string is being read from stdin into eax.

Some code starting at 0016141B does not allow the debugger to step through. We could disable it but it is not required for this assignment.



### Unsuccessful Authentication:

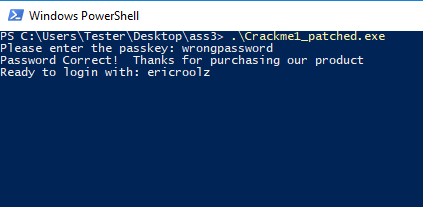
At 00161044 the input string is compared to a string literal “ericroolz”. If the two strings are not equal the code jumps to 00161062 where the code proceeds to jump to 00161083. At this point the failed code pushes the string “Sorry, wrong password.” Onto the stack and calls the address of the printf command to print it.

### Successful Authentication:

At 00161046 the comparison determines that the two strings are equal. It tests if the dl register is equal to itself. If they are equal then it jumps to 0016105E where the eax register is cleared and we jump to 00161067. Since eax was cleared it will be equal to itself and therefore it will not jump. At 0016106B the success string is pushed onto the stack and later printf is called to display it along with the password used.

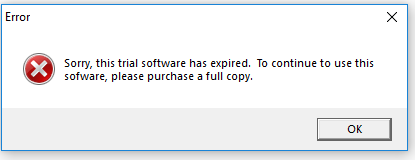
### Patch:

To patch the code, we simply have to replace the JNE instruction at address 00161046 with the instruction “JMP 0016106B” which is the address at which the successful login starts. After applying the patch, we now have a new executable

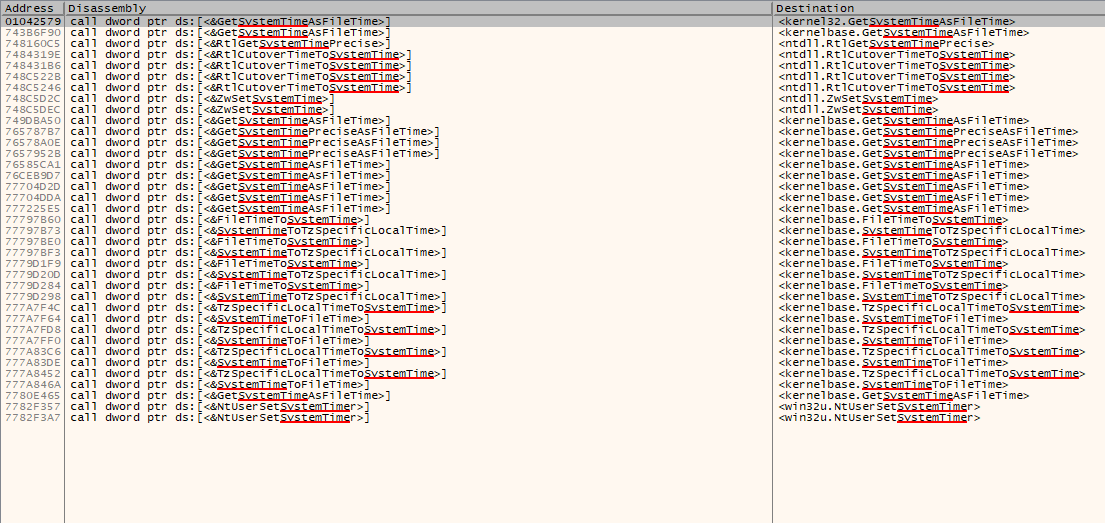


## crackme2.exe

After looking at the function calls of *crackme2* it appears that the application uses a graphical interface as evidenced by the existence of functions such as CreateWindowExW and RegisterClassExW which are core win32 GUI functions. When we run the program, we get this message:



It says that the program has expired, and we can see in the function tables that the program does in fact use many functions which check the current system time:



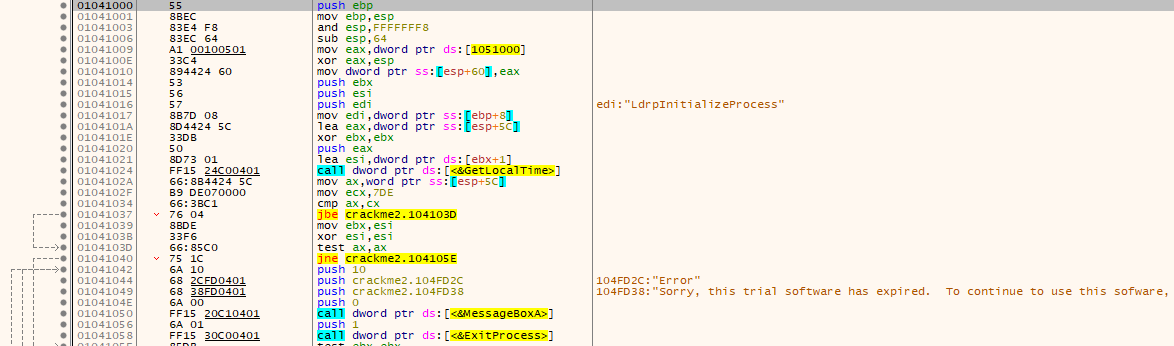
With this information we will now analyze where in the code to see where the string we see in the message box is loaded.

Here we see the strings from the process:





If we follow the string we can see that it is only used right at the top of the executable:

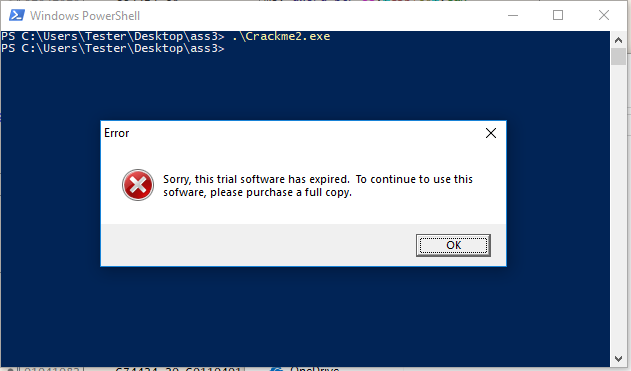


Here we can see that the process calls the win32 function GetLocalTime at address 01041024.

This program appears to not be using a password but instead checking and comparing against a date hardcoded at address 0104102F. This is evidenced by 0x7DE being equal to 2014 in decimal.

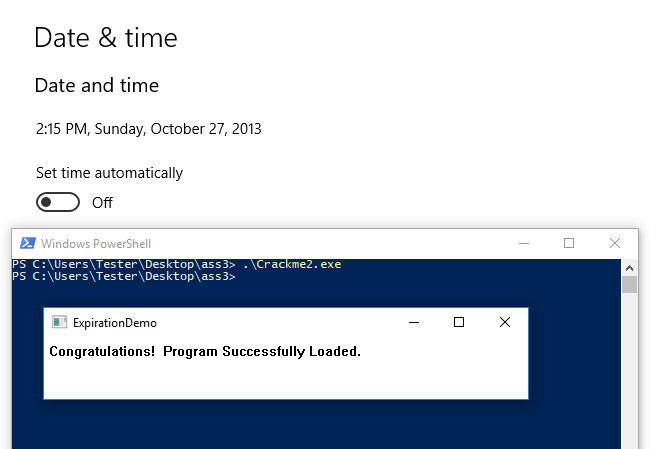
### Unsuccessful Authentication:

We simply execute the code. It checks the date and because our current date is > 2014 we will get the error message.



### Successful Authentication:

Since with the patch we will be modifying the code and the successful authentication requires a less drastic way of getting in we will try to change the current date on our PC.

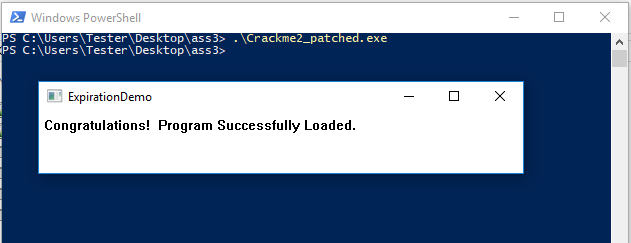


And this works! The code is simple so fooling it by changing the date is trivial.

### Patch:

Since the instruction at 01041037 is checking if AX is below or equal to 2014 we can change the JBE to “JMP 01041066” which will skip all of the code which checks for tampering and or prints error messages.

The output upon running said patched executable is:



# Conclusion

I learned a lot about some of the simpler ways in which developers may try to stop a developer from debugging software. The first program was made to shut down if the presence of a debugger was detected. That could be disabled via a patch, but it didn’t interfere with my analysis, so I kept it. The second piece of code checked against a date which was interesting, but it also used the device’s date which can be modified and should not be trusted.

This was a valuable experience in reverse engineering software with a tool like OllyDbg.