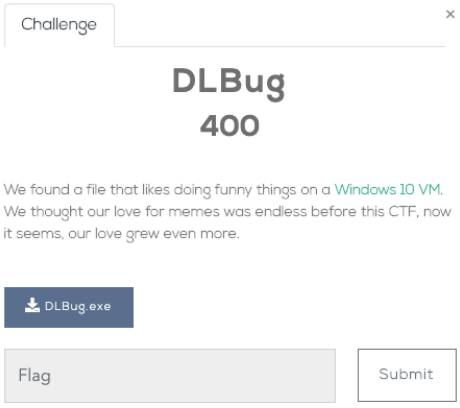
DLBug

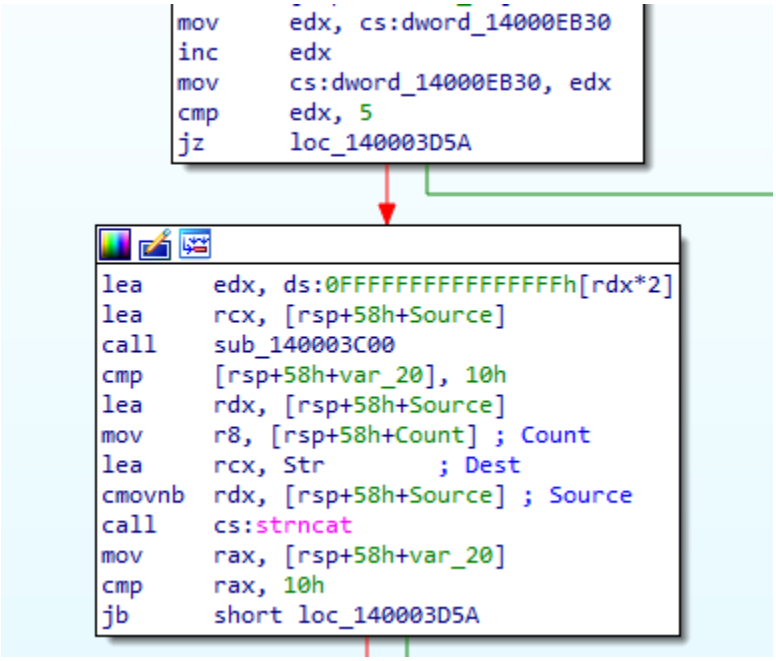
Author: Eran Shimony



You should immediately identify that something is off in this PE file, it contains an additional export point in the form of a *TLS* Callback.

The *TLS* Callback is used for running code before the main thread calls the Main function. The original purpose of this mechanism is to initialize memory and data structure. This method is commonly used by malware authors to run undetected code, a lot more than ordinary software.

In this little piece of code, we can see a conditional statement:



We can see that a global variable under the name dword_14000EB39 stores a counter which is incremented whenever we enter the *TLS* callback. The counter is copied into the edx register, then edx is being compared with the number 5 followed by a jump zero to a code that multiples **EDXs** value and put it in a global array named Str by IDA. It is easy to understand that the copying of **EDX*2** to the array happens no more than four times. The array would store the values "1357", this key phrase will be used later on.

Now, we take a look at the main function. We see there is nothing of interest there. So, we run DLBug.exe and get a message telling us to enter a key. We check for xrefs to the greeting message and we get into a function that seems to be like a global contractor of a class.

```
loc_140003DF0:
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@s
lea     rdx, aEnterYourKey ; "Enter your key: "
mov     [rsp+588h+arg_0], rbx
mov     [rsp+588h+var_8], rdi
call    sub_140005F80
mov     rcx, rax
lea     rdx, sub_140006150
call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAA
mov     rcx, cs:?cin@std@@3V?$basic_istream@DU?$char_traits@D@st
call    sub_1400062E0
lea     rcx, LibFileName ; "ntdll.dll"
call    cs:LoadLibraryA
```

If we put a breakpoint at the beginning of this function we see that the breakpoint is being hit before the main function but after the *TLS* callback.

Before we are requested to enter a key there is an anti-VM check:

```
sub_1400066D0   proc near               ; CODE XREF: sub_140003DC0+19↑p
                xor     eax, eax
                mov     eax, 40000000h
                cpuid
                cmp     ecx, 4D566572h
                jnz     short locret_1400066F0
                cmp     edx, 65726177h
                jnz     short locret_1400066F0
                mov     byte ptr cs:dword_14000EB2C, 1

locret_1400066F0:                       ; CODE XREF: sub_1400066D0+F↑j
                                        ; sub_1400066D0+17↑j
                retn
sub_1400066D0   endp
```

The program checks the virtual processor state if the program runs in a virtual machine the value of **EDX** would be 0x65726177. Therefore, the function would return a value of one. It follows by checking the returned value, exiting the challenge if it is in a VM.

We continue to examine the disassembly, and we see there is a call to LoadLibrary followed by ***GetProcAddress*** to ***NtCreateThreadEx***.

```
call    cs:LoadLibraryA
mov     edx, 400h       ; dwSpinCount
lea     rcx, CriticalSection ; lpCriticalSection
mov     rbx, rax
call    cs:InitializeCriticalSectionAndSpinCount
lea     rdx, ProcName   ; "NtCreateThreadEx"
mov     rcx, rbx        ; hModule
call    cs:GetProcAddress
xor     edi, edi
mov     rbx, rax
mov     [rsp+588h+var_528], rdi
call    cs:GetCurrentProcess
mov     [rsp+588h+var_538], rdi
lea     rcx, [rsp+588h+var_528]
mov     [rsp+588h+var_540], rdi
mov     r9, rax
mov     [rsp+588h+var_548], rdi
lea     rax, sub_140004F60
mov     [rsp+588h+var_550], rdi
xor     r8d, r8d
mov     [rsp+588h+var_558], edi
mov     edx, 10000000h
mov     [rsp+588h+var_560], rdi
mov     [rsp+588h+var_568], rax
call    rbx
```

The **CALL RBX** instruction will activate a thread on sub_140004F560, we put a breakpoint there and continue in the challenge.

A bit after that, there is another anti-VM check. The program checks if the number of CPUs is greater than two, more often than not a user doesn't define the virtual machine to have more than two CPUs, because it leaches CPU time from the host.

```
call    cs:GetSystemInfo
cmp     [rsp+588h+SystemInfo.dwNumberOfProcessors], 3
jnb     short loc_140003F14
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_tra
lea     rdx, aBye       ; "Bye"
call    sub_140005F80
xor     ecx, ecx        ; uExitCode
call    cs:ExitProcess
```

We continue and see there is a code piece that removes all existing hardware breakpoints. This code is executed several times during the program life time. Hardware breakpoints allow the user to break upon reading memory which normally can't be done with software breakpoints. This is another anti-analysis technique that is being used in the challenge.

Afterward, we see there is another call to ***GetProcAddress***, now with the undocumented function name of ***NtSetInformationThread***. This API allows the program to change the data structures inside the *TEB*. It is occasionally used for anti-debug purposes.

```
lea     rcx, LibFileName ; "ntdll.dll"
call    cs:LoadLibraryA
mov     rcx, rax        ; hModule
lea     rdx, aNtsetinformati ; "NtSetInformationThread"
call    cs:GetProcAddress
mov     rbx, rax
call    cs:GetCurrentThread
xor     r9d, r9d
xor     r8d, r8d
mov     rcx, rax
lea     edx, [r9+11h]
call    rbx
```

On x64 calling convention the second argument to a function is passed on the **RDX** register (as long we don't have floating point values). **RDX** would store the value of 0x11 because **R9D** is zeroed with an **XOR** operation, followed by **LEA EDX, [r9+11h]** instruction. This operation would put the value of **R9+0x11** inside **EDX**. The reason behind it is that this argument would cause the program to stop sending notifications to the debugger, effectively terminating the program, because the debugger could not follow the execution.

Finally, we reach the first flag check. Remember that we found that there is a global array being used in the *TLS* callback function? Now the array is being checked to have a length of 4 and followed by a function call with three arguments; a hexadecimal value which looks like a hash value, and two numbers that look like indices of some sort.

```
                mov     edx, 28h ; '('  ; MaxCount
                lea     rcx, Str        ; Str
                call    cs:strnlen
                cmp     rax, 4
                jnb     short loc_140003FAD
                call    sub_1400049E0

loc_140003FAD:                          ; CODE XREF: sub_140003DC0+1E6↑j
                mov     r9d, 4
                xor     r8d, r8d
                mov     rdx, 270163F106DBE9DDh
                call    sub_140004E40
                test    eax, eax
                jnz     short loc_140003FD2
                xor     ecx, ecx        ; uExitCode
                call    cs:ExitProcess
```

The function that is being called, sub_140004E40, is using the ***fnv1a_64*** hash function in order to check if the values of the input [src:dest] that are hashed equal to the value passed on RDX. We feed the string "1357" and meet the condition.

Time for some more Windows Internals, a call to sub_1400048E0 shows some interesting code:

```
xor     rax, rsp
mov     [rsp+38h+var_10], rax
mov     edi, 0FA000h
mov     ecx, edi        ; Size
call    cs:__imp_malloc
xor     ebp, ebp
lea     rcx, LibFileName ; "ntdll.dll"
xor     edx, edx        ; hFile
mov     [rsp+38h+var_18], ebp
mov     r8d, 800h       ; dwFlags
mov     rbx, rax
call    cs:LoadLibraryExA
mov     rcx, rax        ; hModule
lea     rdx, aNtquerysystemi ; "NtQuerySystemInformation"
call    cs:GetProcAddress
lea     r9, [rsp+38h+var_18]
mov     r8d, edi
mov     rsi, rax
lea     ecx, [rbp+5]
mov     rdx, rbx
call    rsi
```

Another call to ***GetProcAddress*** this time with ***NtQuerySystemInformation***. This API is very powerful and allows the caller to get almost any wanted information about the machine, running processes and more. The first argument passed indicates the data structure that would be retuned in the buffer allocated before in **R8**. The value in **ECX** which is the first argument in x64 calling convention is 5 which indicates for ***SystemProcessInformation*** `structure`. The ***NtQuerySystemInformation*** would return an array of pointers that point to a ***SystemProcessInformation*** structure.

Later, we see a loop over this array with a call to sub_14004A80. This function checks if the hashed value under process name [:-4] is 0x617821215C0934C1. If so, it returns the PID of this process. You can see that the returned PID is of a process named SearchUI.exe which would have an important usage in the near future. In addition we can see the hash function that checks if we entered the valid key phrase.

```
loc_140004D74:
mov     r9d, 6
lea     r8d, [r9-2]
mov     rdx, r14
call    sub_140004E90
```

It is pretty clear this is another part of the flag that should be in the final flag. So far we have the flag “1357Search”.

Another thing in this function, there is an interesting check that terminates the program if any of the following programs is running: Windbg, Ida Pro, Procmon and more. It does that by calculating a hash on the process name and comparing it to elements in a vector.

We continue and see the PID of SearchUI.exe is returned in **EAX**

Immediately followed by a function that takes this value as an argument

```
loc_140003FE4:                          ; CODE XREF: sub_140003DC0+219↑j
                call    sub_140004900
                test    eax, eax
                jnz     short loc_140003FF6
                xor     ecx, ecx        ; uExitCode
                call    cs:ExitProcess
; ---------------------------------------------------------------------------
                align 2
; } // starts at 140003DC0

loc_140003FF6:                          ; CODE XREF: sub_140003DC0+22B↑j
                                        ; DATA XREF: .rdata:0000000140008388↓o ...
; __unwind { // __GSHandlerCheck
                mov     [rsp+588h+arg_8], rbp
                mov     edx, eax
                mov     [rsp+588h+arg_10], rsi
                call    sub_1400043C0
                call    sub_1400041E0
```

We go inside the function and see another anti-debug check, the program creates an exception block followed by a ***CloseHandle*** call and the value 0xDEADBEEF, and the debugger will happily catch the exception and prevent the exception handler to deal with it. Skipping the CloseHandle call we can see a copy method into a buffer with the string “G00gle” into another global array.

```
loc_140004348:          ; hObject
;   __try { // __except at loc_14000436D
mov     ecx, 0DEADBEEFh
call    cs:CloseHandle
lea     r8d, [rbx+7]    ; Count
lea     rdx, aG00gle    ; "G00gle"
lea     rcx, Dest       ; Dest
call    cs:strncat
jmp     short loc_140004372
```

The program continues by checking if we entered the “G00gle” string by using the same hash function as before, with a different pair of indices of course.

```
loc_1400044DC:                          ; CODE XREF:
                mov     r9d, 6
                mov     rdx, rbx
                lea     r8d, [r9+4]
                call    sub_140004E90
                test    eax, eax
                jnz     short loc_1400044FB
                xor     ecx, ecx        ; uExitCode
                call    cs:ExitProcess
```

So far we discovered some of the flag: “1357SearchG00gle”. Lets continue to the next piece of code. Here we have a call to ***OpenProcess*** with the PID of SearchUI.exe process, followed by call to ***QueryFullProcessImageNameA*** which should give us the full path of the SearchUI.exe process.

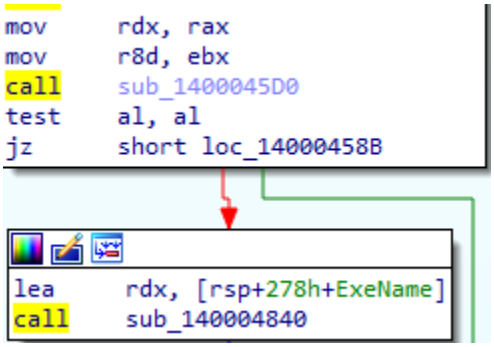
```
mov     r8d, esi
mov     [rsp+278h+dwSize], 208h
xor     edx, edx        ; bInheritHandle
mov     ecx, 400h       ; dwDesiredAccess
call    cs:OpenProcess
lea     r9, [rsp+278h+dwSize] ; lpdwSize
xor     edx, edx        ; dwFlags
mov     rcx, rax        ; hProcess
lea     r8, [rsp+278h+ExeName] ; lpExeName
call    cs:QueryFullProcessImageNameA
```

With that in mind, we see the program tries to load a resource that is inside the PE file of itself. This is done by some API calls to load a resource then putting it in a buffer.

```
lea     r8, Type        ; "Binary"
mov     edx, 65h ; 'e'  ; lpName
xor     ecx, ecx        ; hModule
call    cs:FindResourceA
mov     rdx, rax        ; hResInfo
xor     ecx, ecx        ; hModule
mov     rbx, rax
call    cs:LoadResource
mov     rdx, rbx        ; hResInfo
xor     ecx, ecx        ; hModule
mov     rsi, rax
call    cs:SizeofResource
mov     ebx, eax
test    eax, eax
jz      short loc_14000458B
mov     ecx, ebx        ; Size
call    cs:__imp_malloc
mov     rcx, rsi        ; hResData
mov     rdi, rax
call    cs:LockResource
```

This resource would be used pretty soon. Before that, let's do a small recap. The current key phrases we have is "1357SearchG00gle", in the beginning of this CTF there was function that added a path from appdata to the USER PATH environment variable. A bit after that there is a call to ***NtCreateThreadEx*** that we didn't pay attention to yet.

Ok, now we proceed to the two interesting functions in this challenge, **sub_1400045D0** and **sub_140004840.**



The first one takes the buffer which stores the content of the resource, and the second one takes the path of SearchUI.exe process. The first thing **sub_1400045D0** does is to call ***ExpandEnviromentStrings*** with the *%localappdata"* ***environment*** variable. This API call would give as the full path of %localappdata%; for instance if the username is called *Bob* it would return us the path C:\Users\Bob\Appdata\Local.

```
xor     bl, bl
mov     r8d, 104h       ; nSize
lea     rdx, [rbp+0D0h+Dst] ; lpDst
lea     rcx, Src        ; "%localappdata%"
call    cs:ExpandEnvironmentStringsA
mov     [rsp+1D0h+var_178], 0Fh
xor     r15d, r15d
mov     [rsp+1D0h+var_180], r15
mov     byte ptr [rsp+1D0h+Src], bl
cmp     [rbp+0D0h+Dst], bl
jnz     short loc_140004647
```

So what does the program do with this thing? After it gets the expanded path it joins it with the string *"\DBG\EngineExtenstions"* and then creates a directory with this name if it doesn't exist. After that it writes the resource as a file on this location named *"exts.dll".*

```
lea     rdx, aExtsDll   ; "\\exts.dll"
lea     rcx, [rsp+1D0h+pszPath] ; Src
call    sub_140005D10
mov     [rsp+1D0h+NumberOfBytesWritten], r15d
lea     rcx, [rsp+1D0h+pszPath]
cmp     [rbp+0D0h+var_148], 10h
cmovnb  rcx, [rsp+1D0h+pszPath] ; lpFileName
mov     [rsp+1D0h+hTemplateFile], r15 ; hTemplateFile
mov     [rsp+1D0h+dwFlagsAndAttributes], 80h ; '€' ; dwFlagsAndAttributes
mov     [rsp+1D0h+dwCreationDisposition], 2 ; dwCreationDisposition
xor     r9d, r9d        ; lpSecurityAttributes
xor     r8d, r8d        ; dwShareMode
mov     edx, 40000000h  ; dwDesiredAccess
call    cs:CreateFileA
mov     rdi, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      short loc_1400047AF
```

```
mov     qword ptr [rsp+1D0h+dwCreationDisposition], r15 ; lpOverlapped
lea     r9, [rsp+1D0h+NumberOfBytesWritten] ; lpNumberOfBytesWritten
mov     r8d, esi        ; nNumberOfBytesToWrite
mov     rdx, r14        ; lpBuffer
mov     rcx, rax        ; hFile
call    cs:WriteFile
```

Afterwards, we go in to the second function that got the path of *SearchUI.exe* as an argument. **Sub_140004840** begins by creating a process with the path of *SearchUI.exe* and without any arguments, it waits for it to terminate and returns to the caller.

```
                lea     rax, [rsp+0F8h+ProcessInformation]
                xor     r9d, r9d        ; lpThreadAttributes
                mov     [rsp+0F8h+lpProcessInformation], rax ; lpProcessInformation
                xor     r8d, r8d        ; lpProcessAttributes
                lea     rax, [rsp+0F8h+StartupInfo]
                xor     edx, edx        ; lpCommandLine
                mov     [rsp+0F8h+lpStartupInfo], rax ; lpStartupInfo
                mov     rcx, rbx        ; lpApplicationName
                xor     eax, eax
                mov     [rsp+0F8h+lpCurrentDirectory], rax ; lpCurrentDirectory
                mov     [rsp+0F8h+lpEnvironment], rax ; lpEnvironment
                mov     [rsp+0F8h+dwCreationFlags], eax ; dwCreationFlags
                mov     [rsp+0F8h+bInheritHandles], eax ; bInheritHandles
                call    cs:CreateProcessA
                test    eax, eax
                jz      short loc_1400048D7
                mov     rcx, [rsp+0F8h+ProcessInformation.hProcess] ; hHandle
                or      edx, 0FFFFFFFFh ; dwMilliseconds
                call    cs:WaitForSingleObject
                mov     rcx, [rsp+0F8h+ProcessInformation.hProcess] ; hObject
                call    cs:CloseHandle
                mov     rcx, [rsp+0F8h+ProcessInformation.hThread] ; hObject
                call    cs:CloseHandle

loc_1400048D7:                          ; CODE XREF: sub_140004840+71↑j
                xor     al, al
                mov     rcx, [rsp+0F8h+var_18]
                xor     rcx, rsp        ; StackCookie
                call    __security_check_cookie
                add     rsp, 0F0h
                pop     rbx
                retn
```

Did we miss something? The answer is yes. The creation of SearchUI.exe without any arguments triggers a creation of a process named *WerFault.exe* which is a process responsible to report to Microsoft about errors. So why the program did it in the first place? Let's go back to the ***NtCreateThreadEx*** and the function it calls named **sub_140005560.**

```
xor     edx, edx        ; dwCoInit
xor     ecx, ecx        ; pvReserved
call    cs:CoInitializeEx
xor     r15d, r15d
mov     [rsp+0C0h+pReserved3], r15 ; pReserved3
mov     [rsp+0C0h+dwCapabilities], r15d ; dwCapabilities
mov     [rsp+0C0h+pAuthList], r15 ; pAuthList
mov     [rsp+0C0h+dwImpLevel], 3 ; dwImpLevel
mov     [rsp+0C0h+dwAuthnLevel], r15d ; dwAuthnLevel
xor     r9d, r9d        ; pReserved1
xor     r8d, r8d        ; asAuthSvc
or      esi, 0FFFFFFFFh
mov     edx, esi        ; cAuthSvc
xor     ecx, ecx        ; pSecDesc
call    cs:CoInitializeSecurity
mov     [rbp+57h+ppv], r15
lea     rax, [rbp+57h+ppv]
mov     qword ptr [rsp+0C0h+dwAuthnLevel], rax ; ppv
lea     r9, riid        ; riid
xor     edx, edx        ; pUnkOuter
lea     r8d, [r15+1]    ; dwClsContext
lea     rcx, rclsid     ; rclsid
call    cs:CoCreateInstance
```

It begins with a creation of a COM object, this is a mandatory step if the program wants to talk with the WMI service, spoiler, it does. It continues by creating an intrinsic event, a WMI event that runs after a specific delta.

```
mov     [rdi], rax
mov     [rdi+10h], rax
mov     [rdi+8], r15
mov     dword ptr [rdi+10h], 1
lea     rcx, MultiByteStr ; "SELECT * FROM __InstanceCreationEvent W"...
call    sub_140007330
mov     [rdi], rax
jmp     short loc_1400057B1
```

The query is `"SELECT * FROM __InstanceCreationEvent WITHIN 1 WHERE TargetInstance ISA 'Win32_Process' AND TargetInstance.Name = 'WerFault.exe'"` this query checks if WerFault.exe is created, if so it calls a function named sub_1400034F0, in order to get to this function you must understand how complicated COM objects work or just put a breakpoint when you see the xref to the string “WerFault.exe.log” ☺.

```
mov     edx, 1
lea     r8d, [rdx+3Fh]
lea     rcx, aWerfaultExeLog ; "WerFault.exe.log"
call    cs:?_Fiopen@std@@YAPEAU_iobuf@@PEBDHH@Z ;
test    rax, rax
jz      short loc_1400035A9
```

It tries to open a file named *“WerFault.exe.log”* in the current directory. What is this file? And why it should be there, that is the question. Do you remember that the process **WerFault.exe** was created when the program created the process of *SearchUI.exe?* If you paid attention and used Procmon for monitoring you would see that **WerFault.exe** loads a DLL named **exts.dll**, the buffer which contains the resource.

Let’s disassemble this exts.dll quickly and see what it holds. All the relevant code is placed in the DLLMain and is being called immediately when WerFault loads this DLL file.

```
loc_180001070:
lea     rax, [rbp+12D0h+ProcessInformation]
mov     [rsp+13D0h+lpProcessInformation], rax ; lpProcessInformation
lea     rax, [rbp+12D0h+StartupInfo]
mov     [rsp+13D0h+lpStartupInfo], rax ; lpStartupInfo
mov     [rsp+13D0h+lpCurrentDirectory], rdi ; lpCurrentDirectory
mov     [rsp+13D0h+lpEnvironment], rdi ; lpEnvironment
mov     [rsp+13D0h+dwCreationFlags], edi ; dwCreationFlags
mov     [rsp+13D0h+bInheritHandles], edi ; bInheritHandles
xor     r9d, r9d        ; lpThreadAttributes
xor     r8d, r8d        ; lpProcessAttributes
xor     edx, edx        ; lpCommandLine
lea     rcx, ApplicationName ; "c:\\windows\\system32\\notepad.exe"
call    cs:CreateProcessA
sub     rbx, 1
jnz     short loc_180001070
```

We see that the DLL creates an instance of notepad.exe in a loop, probably just for the lolz. Then there is a call to ***GetModuleFileNameA*** that gets the name of the current process.

```
xor     ecx, ecx        ; hModule
call    cs:GetModuleFileNameA
xor     r9d, r9d        ; fCreate
xor     r8d, r8d        ; csidl
lea     rdx, [rbp+12D0h+pszPath] ; pszPath
xor     ecx, ecx        ; hwnd
call    cs:SHGetSpecialFolderPathA
lea     rax, [rbp+12D0h+Filename]
or      rbx, 0FFFFFFFFFFFFFFFFh
mov     rdx, rbx
```

The returned buffer would be of course WerFault.exe, this is partly the name of the log file we look for. After that the DLL gets the path of the Desktop by using the ***SHGetSpecialFolderPathA*** API. Before the DLL quits it creates a file on the desktop named **werfault.exe.log** with the content "the Golden goose is on the loose".

```
cmp     [rbp+12D0h+var_12C0], 10h
cmovnb  rcx, [rbp+12D0h+var_12D8] ; Filename
lea     rdx, Mode       ; "w"
call    fopen
mov     rbx, rax
mov     r9, rax         ; File
mov     edx, 20h ; ' '  ; Size
lea     r8d, [rdx-1Fh]  ; Count
lea     rcx, aTheGoldenGoose ; "the golden goose is on the loose"
call    fwrite
mov     rcx, rbx        ; File
call    ftell
```

In order to continue in the challenge we need to copy **werfault.exe.log** to the current directory of the challenge's binary and rerun it, because the code that is executed upon the WMI event looks for this very file in the current directory.

We go back to the disassembly of the WMI triggered function and see it looks for the string "Golden" inside the log file, this key phrase would be hashed and compared to the input of the user using the same hash function as before. So the flag looks like this "1357SearchG00gleGolden".

We have little more to uncover, we go back to the global constructer which is our real main function. There is a call to ***sub_1400041E0*** which is the first behind the last interesting function in this program.

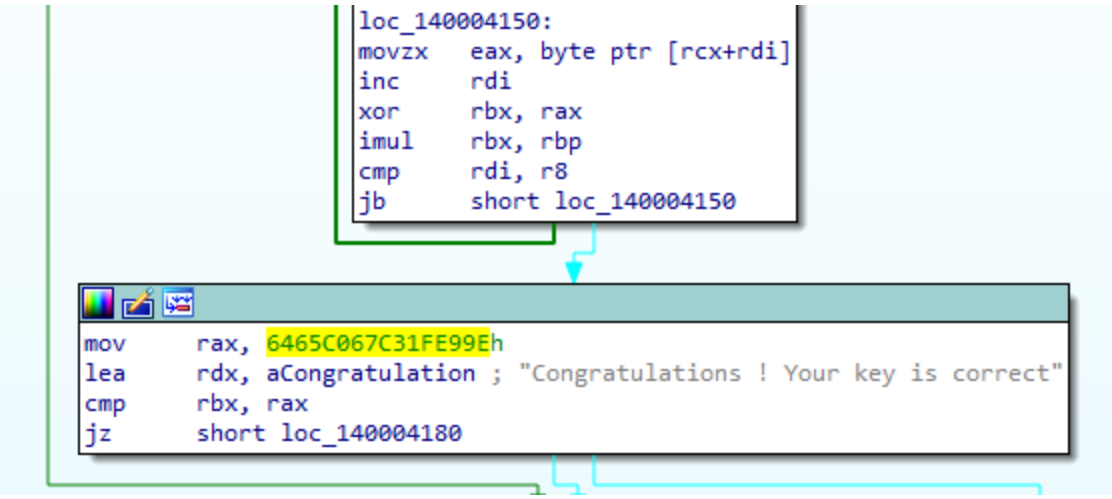
```
lea     rcx, LibFileName ; "ntdll.dll"
mov     cs:byte_14000EB2A, al
call    cs:LoadLibraryExA
mov     rcx, rax         ; hModule
lea     rdx, aNtqueryinforma ; "NtQueryInformationProcess"
call    cs:GetProcAddress
mov     r8d, 3           ; Count
lea     rdx, Source      ; "M3"
lea     rcx, Dest        ; Dest
mov     rbx, rax
call    cs:strncat
call    cs:GetCurrentProcess
mov     r9d, 4
mov     [rsp+528h+var_508], 0
lea     r8, [rsp+528h+var_4F8]
mov     rcx, rax
mov     rdi, rax
lea     edx, [r9+3]
call    rbx
```

Well, we see the char array “M3” is copped into a global array followed by a call to the ***NtQueryInforamtionProcess.*** This API gives any possible information about the process which is identified by the first argument. In this case it is the current process. The second argument in **EDX** register is 0x7 which indicates the **ProcessDebugPort** value inside the **PEB**. Basically, it checks if the debug port is being used, this is another check for anti-debug, almost the last one.

```
mov     r9d, 4
mov     [rsp+528h+var_508], 0
lea     r8, [rsp+528h+var_4F8]
mov     rcx, rdi
lea     edx, [r9+1Bh]
call    rbx
mov     r8d, 3           ; Count
lea     rdx, aM3s        ; "m3s"
lea     rcx, Dest        ; Dest
mov     ebx, eax
call    cs:strncat
```

Well, the last anti-debug method checks if the **PEB** has set the debug flags to one. It is specified by the 0x1F in the second argument. After that it adds to the global array that stores “M3” the string “m3s”. it is clear this string should be combined with the flag we uncovered this far.

We go to the final check with the flag “1357SearchG00gleGoldenM3m3s” in our hand. We face the same hash function as before, comparing out hashed key to the value 0x6465C067C31FE99E.



We get the congratulation message, and we can declare ourselves as victorious.