An Analysis of CheckRemoteDebuggerPresent

Author: Thomas Thelen

Date: 17 October 17, 2015

Motivation

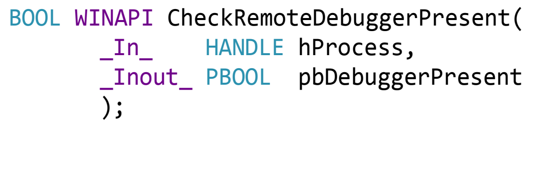
Determining if process if running under a debugger is an important application in software protection and can be used to alter program flow if one is detected. Before attaching a debugger to a remote process it should first be check if the application is already running under a debugger to prevent any problems.

Malware authors also use this technique in order to try to deter the reverse engineer from continuing dynamic analysis. The author may employ tricks such as jumping to a portion of harmless code to appear benign or employ different encryption routines.

The Windows method used to check if a debugger is present in a remote process is by using the Windows API function CheckRemoteDebuggerPresent(). This function can easily be bypassed after taking a closer look at what happens when the function is called and the types of decisions that can be made after.

Theory

Like IsDebuggerPresent, CheckRemoteDebuggerPresent acts as a wrapper for a windows method. Its purpose is to check if the specified process in running under a debugger.

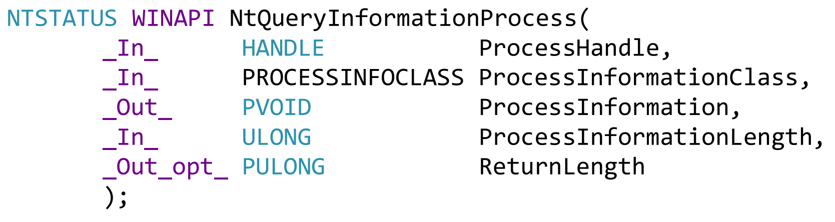


```
BOOL WINAPI CheckRemoteDebuggerPresent(
      _In_    HANDLE hProcess,
      _Inout_ PBOOL  pbDebuggerPresent
      );
```

Figure 1. The MSDN definition of CheckRemoteDebuggerPresent. [1]

The first parameter is a handle to the process which will be checked. To check the current process, it can be a handle to itself. The second parameter is a pointer to a BOOL variable which, in the case of a detected debugger will be set to TRUE and FALSE otherwise [1].

Inside the function, there is a call to the unsupported function NtQueryInformationProcess to retrieve information about the remote process.



```
NTSTATUS WINAPI NtQueryInformationProcess(
      _In_      HANDLE           ProcessHandle,
      _In_      PROCESSINFOCLASS ProcessInformationClass,
      _Out_     PVOID            ProcessInformation,
      _In_      ULONG            ProcessInformationLength,
      _Out_opt_ PULONG           ReturnLength
      );
```

Figure 2. The MSDN entry for NtQueryInformationProcess [2].]

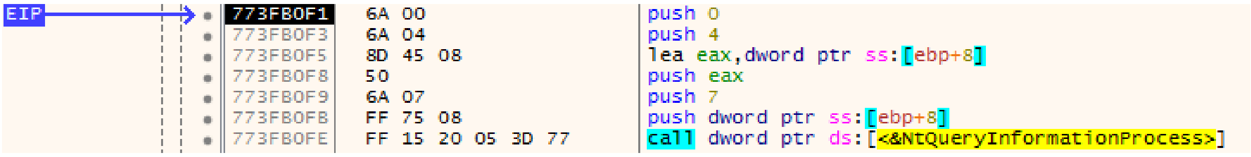
The first parameter is a handle to the process while the second, ProcessInformationClass, is used to specify which information is requested. In order to use this, a handle to the process has to be opened to the process. This can be monitored by hooking kernel32.OpenProcess. If the code resides in an injected DLL, the author may have used kernel32.GetCurrentProcess, which may also be hooked. The third is a pointer to the buffer which stores the requested process information. This will change in size depending on which type of information is requested [2]. The fourth parameter requires the size of the ProcessInformation buffer and the fifth optionally returns a pointer to the actual number of bytes written.

Referring to Figure10, when the ProcessInformationClass is set to seven, it will check if a debugger is present in the process specified by the ProcessHandle. Because an attached debugger will have a port number the buffer will be non-zero in the presence of a debugger.

After the call completes and buffer filled, the eax register is cleared to “0”. The buffer is then checked against eax, saving the result in first eax, and then copying it to the stack. Upon returning from the call, the stack location is read and checked against “1”. From this point it is up to the programmer to decide how program flow should continue.

Disassembly

A disassembly shows the preparation of the stack for the call to NtQueryInformationProcess. Because this is a Win32 call, the calling method is STDCALL. This also means each value's stack size will be a multiple of four and stack cleanup can be expected at the end of the routine.

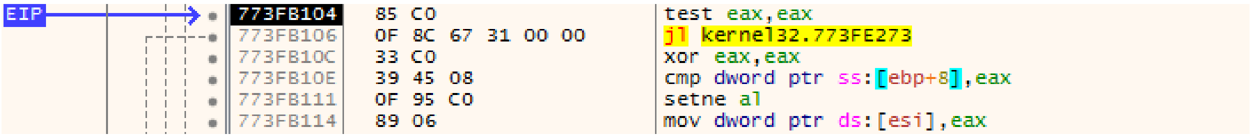


```
EIP ──→ 773FB0F1   6A 00              push 0
        773FB0F3   6A 04              push 4
        773FB0F5   8D 45 08           lea eax,dword ptr ss:[ebp+8]
        773FB0F8   50                 push eax
        773FB0F9   6A 07              push 7
        773FB0FB   FF 75 08           push dword ptr ss:[ebp+8]
        773FB0FE   FF 15 20 05 3D 77  call dword ptr ds:[<&NtQueryInformationProcess>]
```

Figure 3. The preparation of the stack for the call to NtQueryInformationProcess.

The first two instructions respectively set the ReturnLength and ProcessInformationLength. Because ReturnLength is optional, it was given 0. The next instruction loads the buffer's address, ebp+8 into eax. Next, it is pushed on to the stack with ProcessDebugPort (0x07). Finally, the handle is pushed onto the stack. Note that the handle and buffer share the same address space. After the call, the handle is overwritten with the ProcessDebugPort value.

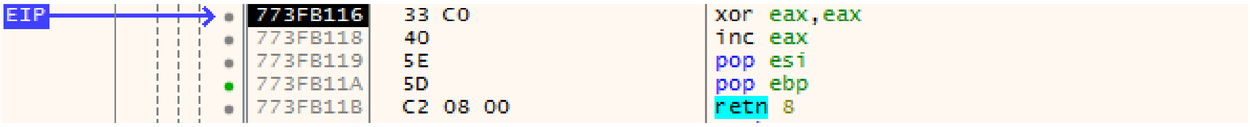
Next, the eax register is cleared by a self xor. The buffer containing the debug information is then checked against eax, which holds "0". The compare will set the zero flag if a debugger is not detected. Immediately after, the eax is set to "1" if the zero flag was not set in the previous instruction. When the debugger is not detected, it remains "0". The value in eax is then written to the data segment pointed to by the esi register.



```
EIP ──→ 773FB104   85 C0              test eax,eax
        773FB106   0F 8C 67 31 00 00  jl kernel32.773FE273
        773FB10C   33 C0              xor eax,eax
        773FB10E   39 45 08           cmp dword ptr ss:[ebp+8],eax
        773FB111   0F 95 C0           setne al
        773FB114   89 06              mov dword ptr ds:[esi],eax
```

Figure 3. Post call operations include writing a "0" or "1" to the stack, depending on the presence of a debugger.

The stack is then cleaned by setting eax to "1" and removing both esi and . The stack location of the variable representing the debugger status is referenced by subtracting from the stack base pointer.

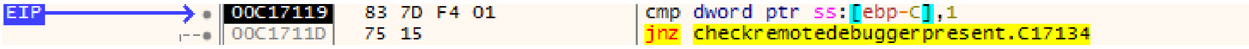


```
EIP ──→ 773FB116   33 C0              xor eax,eax
        773FB118   40                 inc eax
        773FB119   5E                 pop esi
        773FB11A   5D                 pop ebp
        773FB11B   C2 08 00           retn 8
```

Figure 4. The subroutine finishes by cleaning up the stack.

Usage

The byte that was written to the stack after the NtQueryInformationProcess is compared against one. This is checking if a debugger was detected and if so, set the zero flag. In this particular case, the jump is not taken when the debugger is detected. Instead, program flow continues. If the debugger is not detected, the jump will be taken to the normal execution routine.



```
EIP ───────────> ● | 00C17119 | 83 7D F4 01  | cmp dword ptr ss:[ebp-C],1
              ┌--● | 00C1711D | 75 15        | jnz checkremotedebuggerpresent.C17134
```

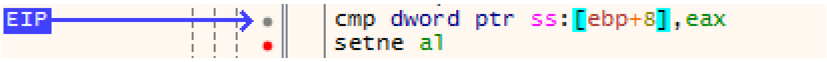
Figure 6. The comparing statement that checks for a detected debugger. If it is detected, it will not take the jump.

Bypassing

Temporary Changes

1. The first way to bypass the check is by setting the zero flag to zero after the compare statement shown in Figure 5.

2. A second way is to change the value of the eax register while inside kernel32.CheckRemoteDebugger. As discussed earlier, the value of ProcessDebugPort was saved in the buffer at ss:[ebp+8]. Before the subroutine returns, it sets the eax register by comparing the buffer with zero.

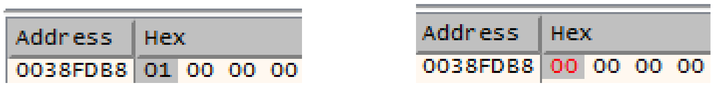


```
EIP   cmp dword ptr ss:[ebp+8],eax
      setne al
```

Figure 5. Eax can be changed on the fly to make this comparison true.

After the comparison is made, the zero flag will not be set because the comparison will never be true if the debugger is detected. The instruction after sets the eax register to the value of the zero flag, in this case 0.

3. The last way to temporarily bypass the check is to manually change the value of the buffer in the stack. Simply changing the "1" to a "0" before the compare will suffice.

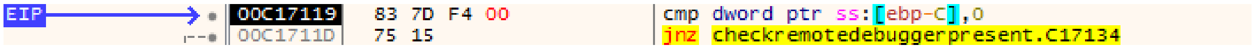


```
Address    Hex                  Address    Hex
0038FDB8   01  00  00  00       0038FDB8   00  00  00  00
```

Figure 6. The buffer before and after it was changed.

Permanent Changes

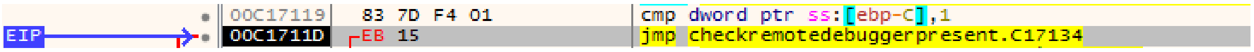
1. Perhaps the easiest way of permanently bypassing the check is by changing the "1" to a "0" in the compare statement. Because the value in ss::[ebp-c] is "1" if a debugger is present, the zero flag is not set. The following jnz instruction is executed because the zero flag was never set.



```
EIP   00C17119   83 7D F4 00   cmp dword ptr ss:[ebp-C],0
      00C1711D   75 15         jnz checkremotedebuggerpresent.C17134
```

Figure 7. Modification of the compare. Note the red 00 where the byte modification was made.

2. Instead of modifying the zero flag, the jnz can be changed to a jmp. This will ensure that the jump to normal code is taken. The size also remains the same so there isn't a need to NOP other instructions.



```
      00C17119   83 7D F4 01   cmp dword ptr ss:[ebp-C],1
EIP   00C1711D   EB 15         jmp checkremotedebuggerpresent.C17134
```

Figure 8. The jnz has been changed to a jmp. This amounts to changing the 75 byte to EB.

Conclusion

CheckRemoteDebugger is different than IsDebuggerPresent in that it has the ability to check any process, given that it has an open handle. Instead of accessing the thread environmental block, it makes a call to NtQueryInformationProcess. This filled a buffer in memory with the value of DEBUG which was non-zero in the case of a detected debugger and “0” otherwise. The value can be read off the stack and checked against “1” or “0”. Program flow can then be directed to via conditional jumps.

The reverser has many options to bypass the check including long term permanent patches and one-time flag changes. The reverser will have more options if she chooses to temporarily bypass the check because it opens up more opportunities for patching in kernel32.CheckRemoteDebugger.

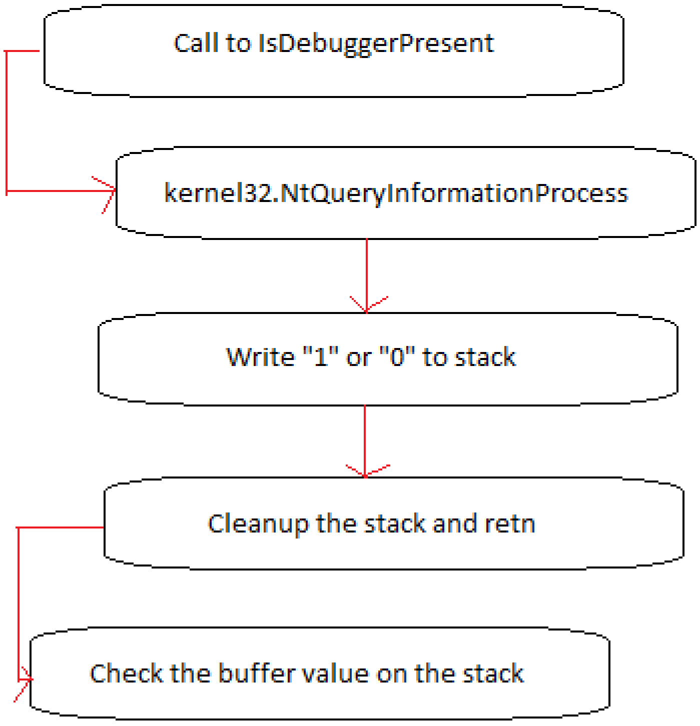

Call to IsDebuggerPresent
kernel32.NtQueryInformationProcess
Write "1" or "0" to stack
Cleanup the stack and retn
Check the buffer value on the stack

Figure 9. The overall flow from the call to IsDebuggerPresent.

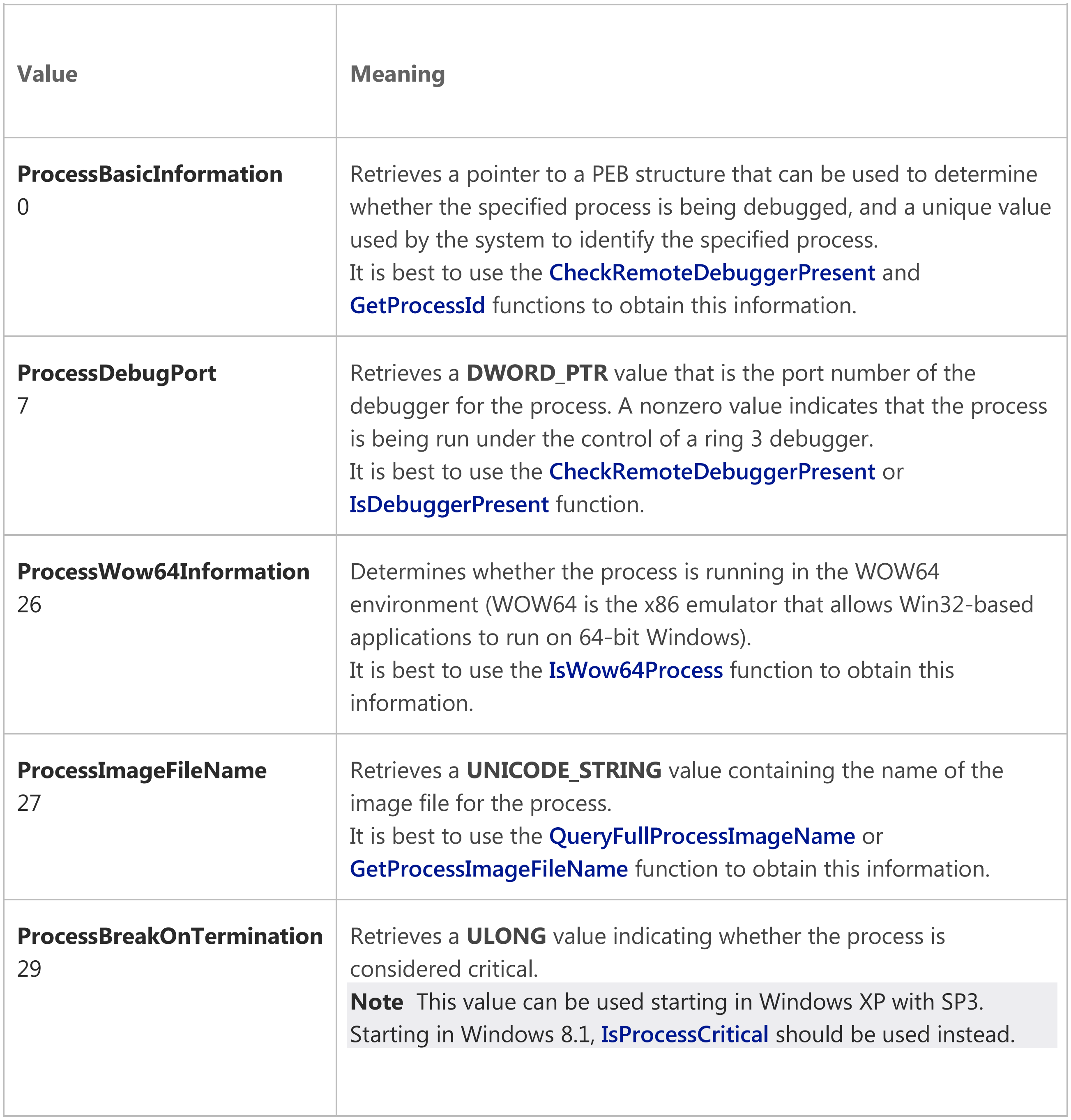
Works Cited

[1] "CheckRemoteDebuggerPresent Function." Microsoft Windows. Web. 17 Oct. 2015. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms679280(v=vs.85).aspx>.

[2] "NtQueryInformationProcess Function." Microsoft Windows. Microsoft. Web. 17 Oct. 2015. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms684280(v=vs.85).aspx>.

[3] "__stdcall." Microsoft Windows. Microsoft, 2015. Web. 17 Oct. 2015. <https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx>.

Additional Figures



Value	Meaning
ProcessBasicInformation 0	Retrieves a pointer to a PEB structure that can be used to determine whether the specified process is being debugged, and a unique value used by the system to identify the specified process. It is best to use the CheckRemoteDebuggerPresent and GetProcessId functions to obtain this information.
ProcessDebugPort 7	Retrieves a **DWORD_PTR** value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger. It is best to use the CheckRemoteDebuggerPresent or IsDebuggerPresent function.
ProcessWow64Information 26	Determines whether the process is running in the WOW64 environment (WOW64 is the x86 emulator that allows Win32-based applications to run on 64-bit Windows). It is best to use the IsWow64Process function to obtain this information.
ProcessImageFileName 27	Retrieves a **UNICODE_STRING** value containing the name of the image file for the process. It is best to use the QueryFullProcessImageName or GetProcessImageFileName function to obtain this information.
ProcessBreakOnTermination 29	Retrieves a **ULONG** value indicating whether the process is considered critical. **Note** This value can be used starting in Windows XP with SP3. Starting in Windows 8.1, IsProcessCritical should be used instead.

Figure 10 Options for the second parameter in NtQueryInformationProcess [2].