The "Ultimate" Anti-Debugging Reference

5/4/2011

Peter Ferrie

Contents

A debugger is probably the most commonly-used tool when reverse-engineering (a disassembler tool such as the Interactive DisAssembler (IDA) being the next most common). As a result, anti-debugging tricks are probably the most common feature of code intended to interfere with reverse-engineering (and anti-disassembly constructs being the next most common). These tricks can simply detect the presence of the debugger, disable the debugger, escape from the control of the debugger, or even exploit a vulnerability in the debugger. The presence of a debugger can be inferred indirectly, or a specific debugger can be detected. Disabling or escaping from the control of the debugger can be achieved in both generic and specific ways. Exploiting a vulnerability, however, is achieved against specific debuggers. Of course, the debugger does not need to be present in order for the exploit to be attempted.

Typically, when a debugger loads, the debuggee's environment is changed by the operating system, to allow the debugger to interact with the debuggee (one exception to this is the Obsidian debugger). Some of these changes are more obvious than others, and affect the operation of the debuggee in different ways. The environment can also be changed in different ways, depending on whether a debugger was used to create a process, or if the debugger attaches to process that is running already.

What follows is a selection of the known techniques used to detect the presence of a debugger, and in some cases, the defences against them.

Note: This text contains a number of code snippets in both 32-bit and 64-bit versions. For simplicity, the 64-bit versions assume that all stack and heap pointers, and all handles, fit in 32 bits. They also rely on the fact that the PEB is always located in low memory.

1.NtGlobalFlag

One of the simplest changes that the system makes is also one of the most misunderstood: the NtGlobalFlag field in the Process Environment Block. The NtGlobalFlag field exists at offset 0x68 in the Process Environment Block on the 32-bit versions of *Windows*, and at offset 0xBC on the 64-bit versions of *Windows*. The value in that field is zero by default. The value is not changed when a debugger attaches to a process. However, the value can be altered to some degree under process control. There are also two registry keys that can be used to set certain values. In the absence of those contributors, a process that is created by a debugger will have a fixed value in this field, by default, but that specific value can be changed by using a certain environment variable. The field is composed of a set of flags. A process that is created by a debugger will have the following flags set:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

Thus, a way to detect the presence of a debugger is to check for the combination of those flags. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
mov eax, fs:[30h] ;Process Environment Block
mov al, [eax+68h] ;NtGlobalFlag
and al, 70h
cmp al, 70h
je  being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
push 60h
pop  rsi
gs:lodsq ;Process Environment Block
```

```
    mov  al, [rsi*2+rax-14h] ;NtGlobalFlag
    and  al, 70h
    cmp  al, 70h
    je   being_debugged
```

Note that for a 32-bit process on the 64-bit versions of *Windows*, there is a separate Process Environment Block for the 32-bit portion and the 64-bit portion. The fields in the 64-bit portion are affected in the same way as for the 32-bit portion.

Thus, there exists another check, which is using this 32-bit code to examine the 64-bit *Windows* environment:

```
    mov eax, fs:[30h] ; Process Environment Block
    ;64-bit Process Environment Block
    ;follows 32-bit Process Environment Block
    mov al, [eax+10bch] ;NtGlobalFlag
    and al, 70h
    cmp al, 70h
    je  being_debugged
```

A common mistake is to use a direct comparison without masking the other bits first. In that case, if any other bits are set, then the presence of the debugger will be missed.

A way to defeat this technique is for the debugger to change the value back to zero before resuming the process. However, as noted above, the initial value can be changed in one of four ways. The first method involves a registry value that affects all processes in the system. That registry value is the "GlobalFlag" string value of the "HKLM\System\CurrentControlSet\Control\Session Manager" registry key. The value here is placed in the NtGlobalFlag field, though it might be changed later by *Windows* (see below). A change to this registry value requires a reboot to take effect. This requirement leads to another way to detect the presence of a debugger which is also aware of the registry value. If the debugger copies the registry value into the NtGlobalFlag field in order to hide its presence, and if the registry value is altered but the system is not rebooted, then the debugger might be fooled into using this new value instead of the true value. The debugger would be revealed if the process knew that the true value was something other than what appears in the registry value. One way to determine the true value would be for the process to run another process, and then query its NtGlobalFlag value. A debugger that is not aware of the registry value is also revealed in this way.

The second method also involves a registry value, but it affects only a named process. That registry value is also the "GlobalFlag" string value, but of the "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>" registry key. The "<filename>" must be replaced by the name of the executable file (not a DLL) to which the flags will be applied when the file is executed. As above, the value here is placed in the NtGlobalFlag field, though it might be changed later by *Windows* (see below). The value that is set using this method is merged with the value that is applied to all processes, if present.

The third method to change the value relies on two fields in the Load Configuration Table. One field (GlobalFlagsClear) lists the flags to clear, and the other field (GlobalFlagsSet) lists the flags to set. These settings are applied after the GlobalFlag registry value(s) has/have been applied, so they can override the values specified in the GlobalFlag registry value(s). However, they cannot override the values that *Windows* sets when certain flags remain set (though they can remove the flags that are set when a debugger creates a process). For example, setting the FLG_USER_STACK_TRACE_DB (0x1000) flag causes *Windows* to set the FLG_HEAP_VALIDATE_PARAMETERS (0x40) flag. If the FLG_USER_STACK_TRACE_DB flag is set in either of the GlobalFlag registry values, then even if the FLG_HEAP_VALIDATE_PARAMETERS flag is marked for clearing in the Load Configuration Table, it will still be set by *Windows* later during the process load.

The fourth method is specific to the changes that *Windows* makes when a debugger creates a process. By setting the "_NO_DEBUG_HEAP" environment variable, the three heap flags will not be set in the NtGlobalFlag field because of the debugger. They can, of course, still be set by the GlobalFlag registry values or the GlobalFlagsSet field in the Load Configuration Table.

2.Heap flags

The heap contains two flags that are initialised in conjunction with the NtGlobalFlag. The values in those fields are affected by the presence of a debugger, but also depend on the version of *Windows*. The location of those fields depends on the version of *Windows*. The two fields are named "Flags" and "ForceFlags". The Flags field exists at offset 0x0C in the heap on the 32-bit versions of *Windows NT*, *Windows 2000*, and *Windows XP*; and at offset 0x40 on the 32-bit versions of *Windows Vista* and later. The Flags field exists at offset 0x14 in the heap on the 64-bit versions of *Windows XP*, and at offset 0x70 in the heap on the 64-bit versions of *Windows Vista* and later. The ForceFlags field exists at offset 0x10 in the heap on the 32-bit versions of *Windows NT*, *Windows 2000*, and *Windows XP*; and at offset 0x44 on the 32-bit versions of *Windows Vista* and later. The ForceFlags field exists at offset 0x18 in the heap on the 64-bit versions of *Windows XP*, and at offset 0x74 in the heap on the 64-bit versions of *Windows Vista* and later.

The value for the Flags field is normally set to HEAP_GROWABLE (2) on all versions of *Windows*. The value for the ForceFlags field is normally set to zero on all versions of *Windows*. However, both of these values depend on the subsystem version of the host process, for a 32-bit process (a 64-bit process has no such dependency). The field values are as stated only if the subsystem version is 3.51 or greater. If the subsystem version is 3.10-3.50, then the HEAP_CREATE_ALIGN_16 (0x10000) flag will also be set in both fields. If the subsystem version is less than 3.10, then the file will not run at all. This is especially interesting because a common technique is to place the two and zero values in their respective fields, in order to hide the presence of the debugger. However, if the subsystem version is not checked, then that action might reveal the presence of something that is attempting to hide the debugger.

When a debugger is present, the Flags field is normally set to the combination of these flags on *Windows NT*, *Windows 2000*, and 32-bit *Windows XP*:

```
HEAP_GROWABLE (2)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
```

On 64-bit *Windows XP*, and *Windows Vista* and later, the Flags field is normally set to the combination of these flags:

```
HEAP_GROWABLE (2)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
```

When a debugger is present, the ForceFlags field is normally set to the combination of these flags:

```
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
```

The HEAP_TAIL_CHECKING_ENABLED flag is set in the heap fields if the FLG_HEAP_ENABLE_TAIL_CHECK flag is set in the NtGlobalFlag field. The HEAP_FREE_CHECKING_ENABLED flag is set in the heap fields if the FLG_HEAP_ENABLE_FREE_CHECK flag is set in the NtGlobalFlag field. The HEAP_VALIDATE_PARAMETERS_ENABLED flag (and the HEAP_CREATE_ALIGN_16 (0x10000) flag on *Windows NT* and *Windows 2000*) is set in the heap fields if the

FLG_HEAP_VALIDATE_PARAMETERS flag is set in the NtGlobalFlag field.

This behaviour can be prevented on *Windows XP* and later, causing the default values to be used instead, by creating the environment variable "_NO_DEBUG_HEAP".

The heap flags can also be controlled on a per-process basis, through the "PageHeapFlags" string value of the "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>" registry key.

The location of the heap can be retrieved in several ways. One way is by using the kernel32 GetProcessHeap() function. It is equivalent to using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov eax, fs:[30h] ;Process Environment Block
    mov eax, [eax+18h] ;get process heap base
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    push 60h
    pop  rsi
    gs:lodsq ;Process Environment Block
    mov  eax, [rax+30h] ;get process heap base
```

As with the Process Environment Block, for a 32-bit process on the 64-bit versions of *Windows*, there is a separate heap for the 32-bit portion and the 64-bit portion. The fields in the 64-bit portion are affected in the same way as for the 32-bit portion.

Thus, there exists another check, which is using this 32-bit code to examine the 64-bit *Windows* environment:

```
    mov eax, fs:[30h] ;Process Environment Block
    ;64-bit Process Environment Block
    ;follows 32-bit Process Environment Block
    mov eax, [eax+1030h] ;get process heap base
```

Another way is by using the kernel32 GetProcessHeaps() function. This function is simply forwarded to the ntdll RtlGetProcessHeaps() function. The function returns an array of the process heaps. The first heap in the list is the same as the one returned by the kernel32 GetProcessHeap() function. The query can also be performed using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push 30h
pop  esi
fs:lodsd ;Process Environment Block
;get process heaps list base
mov  esi, [esi+eax+5ch]
lodsd
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
push 60h
pop  rsi
gs:lodsq ;Process Environment Block
;get process heaps list base
mov  esi, [rsi*2+rax+20h]
lodsd
```

or using this 32-bit code to examine the 64-bit *Windows* environment:

```
mov eax, fs:[30h] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov esi, [eax+10f0h] ;get process heaps list base
lodsd
```

Thus, a way to detect the presence of a debugger is to check for the special combination of flags in the Flags field. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is (or might be) within the 3.10-3.50 range:

```
    call GetVersion
    cmp  al, 6
    cmc
    sbb  ebx, ebx
    and  ebx, 34h
    mov  eax, fs:[30h] ;Process Environment Block
    mov  eax, [eax+18h] ;get process heap base
    mov  eax, [eax+ebx+0ch] ;Flags
    ;neither HEAP_CREATE_ALIGN_16
    ;nor HEAP_SKIP_VALIDATION_CHECKS
    and  eax, 0effeffffh
    ;HEAP_GROWABLE
    ;+ HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    cmp  eax, 40000062h
    je   being_debugged
```

or this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is 3.51 or greater:

```
    call  GetVersion
    cmp   al, 6
    cmc
    sbb   ebx, ebx
    and   ebx, 34h
    mov   eax, fs:[30h] ;Process Environment Block
    mov   eax, [eax+18h] ;get process heap base
    mov   eax, [eax+ebx+0ch] ;Flags
    ;not HEAP_SKIP_VALIDATION_CHECKS
    bswap eax
    and   al, 0efh
    ;HEAP_GROWABLE
    ;+ HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    ;reversed by bswap
    cmp  eax, 62000040h
    je   being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
push 60h
pop  rsi
gs:lodsq ;Process Environment Block
mov  ebx, [rax+30h] ;get process heap base
call GetVersion
cmp  al, 6
sbb  rax, rax
and  al, 0a4h
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp  d [rbx+rax+70h], 40000062h ;Flags
je   being_debugged
```

or using this 32-bit code to examine the 64-bit *Windows* environment:

```
push 30h
pop  eax
mov  ebx, fs:[eax] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov  ah, 10h
mov  ebx, [ebx+eax] ;get process heap base
call GetVersion
cmp  al, 6
sbb  eax, eax
and  al, 0a4h
;Flags
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp  [ebx+eax+70h], 40000062h
je   being_debugged
```

The kernel32 GetVersion() function call can be further obfuscated by simply retrieving the value directly from the NtMajorVersion field in the KUSER_SHARED_DATA structure, at offset 0x7ffe026c for 2Gb user-space configurations. This value is available on all 32-bit and 64-bit versions of *Windows*.

Another way to detect the presence of a debugger is to check for the special combination of flags in the ForceFlags field. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is (or might be) within the 3.10-3.50 range:

```
    call GetVersion
    cmp  al, 6
    cmc
    sbb  ebx, ebx
    and  ebx, 34h
    mov  eax, fs:[30h] ;Process Environment Block
    mov  eax, [eax+18h] ;get process heap base
    mov  eax, [eax+ebx+10h] ;ForceFlags
    ;not HEAP_CREATE_ALIGN_16
    btr  eax, 10h
    ;HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    cmp  eax, 40000060h
    je   being_debugged
```

or using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is 3.51 or greater:

```
    call GetVersion
    cmp  al, 6
    cmc
    sbb  ebx, ebx
    and  ebx, 34h
    mov  eax, fs:[30h] ;Process Environment Block
    mov  eax, [eax+18h] ;get process heap base
    ;ForceFlags
    ;HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    cmp  [eax+ebx+10h], 40000060h
    je   being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    push 60h
    pop  rsi
    gs:lodsq ;Process Environment Block
    mov  ebx, [rax+30h] ;get process heap base
    call GetVersion
    cmp  al, 6
    sbb  rax, rax
    and  al, 0a4h
    ;ForceFlags
    ;HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    cmp  d [rbx+rax+74h], 40000060h
    je   being_debugged
```

or using this 32-bit code to examine the 64-bit *Windows* environment:

```
    call GetVersion
    cmp  al, 6
    push 30h
    pop  eax
    mov  ebx, fs:[eax] ;Process Environment Block
    ;64-bit Process Environment Block
    ;follows 32-bit Process Environment Block
    mov  ah, 10h
    mov  ebx, [ebx+eax] ;get process heap base
    sbb  eax, eax
    and  al, 0a4h
    ;ForceFlags
    ;HEAP_TAIL_CHECKING_ENABLED
    ;+ HEAP_FREE_CHECKING_ENABLED
    ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
    cmp  [ebx+eax+74h], 40000060h
    je   being_debugged
```

3.The Heap

When the heap is initialised, the heap flags are checked, and depending on which flags are set, there might be additional changes to the environment. If the HEAP_TAIL_CHECKING_ENABLED flag is set, then the sequence 0xABABABAB will be appended twice in a 32-bit *Windows* environment (four times in a 64-bit *Windows* environment) at the exact end of the allocated block. If the HEAP_FREE_CHECKING_ENABLED flag is set, then the sequence 0xFEEEFEEE (or a part thereof) will be appended if additional bytes are required to fill in the slack space until the next block. Thus, a way to detect the presence of a debugger is to check for those values. If a heap pointer is known, then the check can be made by examining the heap data directly. However, *Windows Vista* and later use heap protection on both the 32-bit and 64-bit platforms, with the introduction of an XOR key to encode the block size. The use of this key is optional, but by default it is used. The location of the overhead field is also different between *Windows NT/2000/XP* and *Windows Vista* and later. Therefore, the version of *Windows* must be taken into account. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor    ebx, ebx
    call   GetVersion
    cmp    al, 6
    sbb    ebp, ebp
    jb     l1
    ;Process Environment Block
    mov    eax, fs:[ebx+30h]
    mov    eax, [eax+18h] ;get process heap base
    mov    ecx, [eax+24h] ;check for protected heap
    jecxz  l1
    mov    ecx, [ecx]
    test   [eax+4ch], ecx
    cmovne ebx, [eax+50h] ;conditionally get heap key
l1: mov    eax, <heap ptr>
    movzx  edx, w [eax-8] ;size
    xor    dx, bx
    movzx  ecx, b [eax+ebp-1] ;overhead
    sub    eax, ecx
    lea    edi, [edx*8+eax]
    mov    al, 0abh
    mov    cl, 8
```

```
    repe   scasb
    je     being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor    ebx, ebx
    call   GetVersion
    cmp    al, 6
    sbb    rbp, rbp
    jb     l1
    ;Process Environment Block
    mov    rax, gs:[rbx+60h]
    mov    eax, [rax+30h] ;get process heap base
    mov    ecx, [rax+40h] ;check for protected heap
    jrcxz  l1
    mov    ecx, [rcx+8]
    test   [rax+7ch], ecx
    cmovne ebx, [rax+88h] ;conditionally get heap key
l1: mov    eax, <heap ptr>
    movzx  edx, w [rax-8] ;size
    xor    dx, bx
    add    edx, edx
    movzx  ecx, b [rax+rbp-1] ;overhead
    sub    eax, ecx
    lea    edi, [rdx*8+rax]
    mov    al, 0abh
    mov    cl, 10h
    repe   scasb
    je     being_debugged
```

There is no equivalent for 32-bit code to examine the 64-bit *Windows* environment because the 64-bit heap cannot be parsed by the 32-bit heap function.

If no pointer is known, then one can be retrieved by using the kernel32 HeapWalk() or the ntdll RtlWalkHeap() function (or even the kernel32 GetCommandLine() function). The returned block size value is decoded automatically, so the version of *Windows* no longer matters in this case. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov   ebx, offset l2
    ;get a pointer to a heap block
l1: push  ebx
    mov   eax, fs:[30h] ;Process Environment Block
    push d [eax+18h] ;save process heap base
    call HeapWalk
    cmp  w [ebx+0ah], 4 ;find allocated block
    jne  l1
    mov  edi, [ebx] ;data pointer
    add  edi, [ebx+4] ;data size
    mov  al, 0abh
    push 8
    pop  ecx
    repe scasb
    je   being_debugged
    ...
l2: db   1ch dup (0) ;sizeof(PROCESS_HEAP_ENTRY)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rbx, offset l2
    ;get a pointer to a heap block
l1: push rbx
    pop  rdx
    push 60h
    pop  rsi
    gs:lodsq ;Process Environment Block
    ;get a pointer to process heap base
    mov  ecx, [rax+30h]
    call HeapWalk
    cmp  w [rbx+0eh], 4 ;find allocated block
    jne  l1
    mov  edi, [rbx] ;data pointer
    add  edi, [rbx+8] ;data size
    mov  al, 0abh
    push 10h
    pop  rcx
    repe scasb
    je   being_debugged
    ...
l2: db   28h dup (0) ;sizeof(PROCESS_HEAP_ENTRY)
```

There is no equivalent for 32-bit code to examine the 64-bit *Windows* environment because the 64-bit heap cannot be parsed by the 32-bit heap function.

4.Thread Local Storage

Thread Local Storage is one of the most interesting anti-debugging techniques that exist, because despite being known for more than ten years, new ways are still being discovered to use (and abuse) it. Thread Local Storage exists to initialise thread-specific data before that thread runs. Since every process contains at least one thread, that behaviour includes the ability to initialise data before the main thread runs. The initialisation can be done by specifying a static buffer that is copied to dynamically allocated memory, and/or via the execution of code in an array of callbacks, to initialise memory contents dynamically. It is the callback array that is abused most often.

The Thread Local Storage callback array can be altered (later entries can be modified) and/or extended (new entries can be appended) at runtime. Newly added or modified callbacks will be called, using the new addresses. There is no limit to the number of callbacks that can be placed. The extension can be made using this code (identical for 32-bit and 64-bit) on either the 32-bit or 64-bit versions of *Windows*:

```
l1: mov d [offset cbEnd], offset l2
    ret
l2: ...
```

The callback at l2 will be called when the callback at l1 returns.

Thread Local Storage callback addresses can point outside of the image, for example, to newly loaded DLLs. This can be done indirectly, by loading the DLL and placing the returned address into the Thread Local Storage callback array. It can also be done directly, if the loading address of the DLL is known. The imagebase value can be used as the callback address, if the DLL is structured in such a way as to defeat Data Execution Prevention, in case it is enabled, or a valid export address can be retrieved and used. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
l1: push offset l2
    call LoadLibraryA
    mov  [offset cbEnd], eax
    ret
l2: db   "myfile", 0
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  rcx, offset l2
    call LoadLibraryA
    mov  [offset cbEnd], rax
    ret
l2: db   "myfile", 0
```

In this case, the "MZ" header of the file named "tls2.dll" will be executed when the callback at l1 returns. Alternatively, the file could refer to itself using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
l1: push 0
    call GetModuleHandleA
    mov  [offset cbEnd], eax
    ret
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
l1: xor  ecx, ecx
    call GetModuleHandleA
    mov  [offset cbEnd], rax
    ret
```

In this case, the "MZ" header of the current process will be executed when the callback at l1 returns.

Thread Local Storage callback addresses can contain RVAs of imported addresses from other DLLs, if the import address table is altered to point into the callback array. Imports are resolved before callbacks are called, so imported functions will be called normally when the callback array entry is reached.

Thread Local Storage callbacks receive three stack parameters, which can be passed directly to functions. The first parameter is the ImageBase of the host process. It could be used by the kernel32 LoadLibrary() function or kernel32 WinExec() function, for example. The ImageBase parameter will be interpreted by the kernel32 LoadLibrary() or kernel32 WinExec() functions as a pointer to the file name to load or execute. By creating a file called "MZ[some string]", where "[some string]" matches the host file header contents, the Thread Local Storage callback will access the file without any explicit reference. Of course, the "MZ" portion of the string can also be replaced manually at runtime, but many functions rely on this signature, so the results of such a change are unpredictable.

Thread Local Storage callbacks are called whenever a thread is created or destroyed (unless the process calls the kernel32 DisableThreadLibraryCalls() or the ntdll LdrDisableThreadCalloutsForDll() functions). That includes the thread that is created by Windows when a debugger attaches to a process. The debugger thread is special, in that its entrypoint does not point inside the image. Instead, it points inside kernel32.dll. Thus, a simple debugger detection method is to use a Thread Local Storage callback to query the start address of each thread that is created. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push eax
    mov  eax, esp
    push 0
    push 4
    push eax
```

```
    ;ThreadQuerySetWin32StartAddress
    push 9
    push -2 ;GetCurrentThread()
    call NtQueryInformationThread
    pop  eax
    cmp  eax, offset l1
    jnb  being_debugged
    ...
l1: <code end>
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   ebp, ebp
    enter 20h, 0
    push  4
    pop   r9
    push  rbp
    pop   r8
    ;ThreadQuerySetWin32StartAddress
    push  9
    pop   rdx
    push  -2 ;GetCurrentThread()
    pop   rcx
    call  NtQueryInformationThread
    leave
    cmp   rbp, offset l1
    jnb   being_debugged
    ...
l1: <code end>
```

Since Thread Local Storage callbacks run before a debugger can gain control, the callback can make other changes, such as removing the breakpoint that is typically placed at the host entrypoint. The patch can be made using this code (identical for 32-bit and 64-bit) on either the 32-bit or 64-bit versions of *Windows*:

```
    ;<val> is byte at l1
    mov  b [offset l1], <val>
    ret
l1: <host entrypoint>
```

The defence against this technique is very simple, and increasingly necessary. It is a matter of inserting the breakpoint on the first byte of the first Thread Local Storage callback, instead of at the host entrypoint. This will allow the debugger to gain control before any code can run in the process (excluding any loaded DLLs, of course). However, care must be taken regarding the callback address, since as noted above the original value at that address can be the RVA of an imported function. Thus, the address cannot be read from the file. It must be read from the image memory.

The execution of Thread Local Storage callbacks is also platform-specific. If the executable imports only from either ntdll.dll or kernel32.dll, then callbacks will not be called during the "on attach" event when run on *Windows XP* and later. When a process starts, the ntdll LdrInitializeThunk() function processes the InLoadOrderModuleList list. The InLoadOrderModuleList list contains the list of DLLs to process. The Flags value in the referenced structure must have the LDRP_ENTRY_PROCESSED bit clear in at least one DLL for the Thread Local Storage callbacks to be called on attach.

That bit is always set for ntdll.dll, so a file importing from only ntdll.dll will not have Thread Local Storage callbacks executed on attach. *Windows 2000* and earlier had a crash bug if a file did not import from kernel32.dll, either explicitly (that is, importing from kernel32.dll directly) or implicitly (that is, importing from a DLL that imports from kernel32.dll; or a DLL that imports from … a DLL that imports from kernel32.dll, regardless of how long the chain is).

This bug was fixed in *Windows XP*, by forcing ntdll.dll to explicitly load kernel32.dll, before processing the host import table. When kernel32.dll is loaded, it is added to the InLoadOrderModuleList. The problem is that this fix introduced a side-effect.

The side-effect occurs when ntdll.dll retrieves an exported function address from kernel32.dll, via the ntdll LdrGetProcedureAddressEx() function. The side-effect would be triggered as a result of retrieving any exported function, but it is triggered in this particular case by ntdll retrieving the address of one of the following functions: BaseProcessInitPostImport() (*Windows XP and Windows Server 2003* only), BaseQueryModuleData() (*Windows XP and Windows Server 2003* only, if the BaseProcessInitPostImport() function does not exist), BaseThreadInitThunk() (*Windows Vista* and later versions), or BaseQueryModuleData() (*Windows Vista* and later versions, if BaseThreadInitThunk() does not exist).

The side-effect is that the ntdll LdrGetProcedureAddressEx() function sets the LDRP_ENTRY_PROCESSED flag for the kernel32.dll entry in the InLoadOrderModuleList list. As a result, a file importing from only kernel32.dll will no longer have Thread Local Storage callbacks executed on attach. This could be considered a bug in *Windows*.

There is a simple workaround for the problem, which is to import something from another DLL, and provided that the DLL has a non-zero entrypoint. Then the Thread Local Storage callbacks will be executed on attach. The workaround works because the Flags field value will have the LDRP_ENTRY_PROCESSED bit clear for that DLL.

On *Windows Vista* and later, dynamically-loaded DLLs also support Thread Local Storage. This is in direct contradiction to the existing Portable Executable format documentation, which states that "Statically declared TLS data objects", that is to say, Thread Local Storage callbacks, "can be used only in statically loaded image files. This fact makes it unreliable to use static Thread Local Storage data in a DLL unless you know that the DLL, or anything statically linked with it, will never be loaded dynamically with the LoadLibrary API function". Further, the Thread Local Storage callbacks will be called, no matter what is present in the import table. Thus, the DLL can import from ntdll.dll or kernel32.dll or even no DLLs at all (unlike the .exe case described above), and the callbacks will be called!

5.Anti-Step-Over

Most debuggers support stepping over certain instructions, such as "call" and "rep" sequences. In such cases, a software breakpoint is often placed in the instruction stream, and then the process is allowed to resume execution. The debugger normally receives control again when the software breakpoint is reached. However, in the case of the "rep" sequence, the debugger must check that the instruction following the rep prefix is indeed an instruction to which the rep applies legally. Some debuggers assume that any rep prefix precedes a string instruction. This introduces a vulnerability when the instruction following the rep prefix is another instruction entirely. Specifically, the problem occurs if that instruction removes the software breakpoint that would be placed in the stream if the instruction were stepped over. In that case, when the instruction is stepped over, and the software breakpoint is removed by the instruction, execution resumes under complete control of the process and never returns to the debugger.
Example code looks like this:

```
    rep
l1: mov b [offset l1], 90h
l2: nop
```

If a step-over is attempted at l1, then execution will resume freely from l2.

A more generic method uses the string instructions to remove the breakpoint. The patch can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    mov al, 90h
    xor ecx, ecx
    inc ecx
    mov edi, offset l1
    rep stosb
l1: nop
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov al, 90h
    xor ecx, ecx
    inc ecx
    mov rdi, offset l1
    rep stosb
l1: nop
```

There are variations of this technique, such as using "rep movs" instead of "rep stos". The direction flag can be used to reverse the direction of the memory write, so that the overwrite might be overlooked. The patch can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  al, 90h
    push 2
    pop  ecx
    mov  edi, offset l1
    std
    rep  stosb
    nop
l1: nop
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  al, 90h
    push 2
    pop  rcx
    mov  rdi, offset l1
    std
    rep  stosb
    nop
l1: nop
```

The solution to this problem is to use hardware breakpoints during step-over of string instructions. This is especially important when one considers that the debugger has no way of knowing if the breakpoint that it placed is the breakpoint that executed. If a process removes the breakpoint, it can also restore the breakpoint afterwards, and then execute the breakpoint as usual. The debugger will see the breakpoint exception that it was expecting, and behave as normal. The ruse can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    mov al, 90h
l1: xor ecx, ecx
    inc ecx
    mov edi, offset l3
l2: rep stosb
l3: nop
    cmp al, 0cch
l4: mov al, 0cch
    jne l1
l5: ...
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov al, 90h
l1: xor ecx, ecx
    inc ecx
    mov rdi, offset l3
l2: rep stosb
l3: nop
    cmp al, 0cch
l4: mov al, 0cch
    jne l1
l5: ...
```

In this example, stepping over the instruction at l2 will allow the code to reach l4, and then return to l1. This will cause the breakpoint to be replaced by l2 on the second pass, and executed by l3. The debugger will then regain control. At that time, the only obvious difference will be that the AL register will hold the value 0xCC instead of the expected 0x90. This will allow l5 to be reached in what appears to be one pass instead of two. Of course, much more subtle variations are possible, including the execution of entirely different code-paths.

A variation of the technique can be used to simply detect the presence of a debugger. The check can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor ecx, ecx
    inc ecx
    mov esi, offset l1
    lea edi, [esi + 1]
    rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
    je  being_debugged
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    xor ecx, ecx
    inc ecx
    mov rsi, offset l1
    lea rdi, [esi + 1]
    rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
    je  being_debugged
```

This code will detect a breakpoint that is placed at l1. It works by copying the value at l1 over the "90h" at l1+1. The value is then compared at l2.

6.Hardware

A.Hardware breakpoints

When an exception occurs, *Windows* creates a context structure to pass to the exception handler. The structure will contain the values of the general registers, selectors, control registers, and the debug registers. If a debugger is present and passes the exception to the debuggee with hardware breakpoints in use, then the debug registers will contain values that reveal the presence of the debugger. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    int  3 ;force an exception to occur
    ...
l1: ;execution resumes here when exception occurs
    mov  eax, [esp+0ch] ;get ContextRecord
    mov  ecx, [eax+4] ;Dr0
    or   ecx, [eax+8] ;Dr1
    or   ecx, [eax+0ch] ;Dr2
    or   ecx, [eax+10h] ;Dr3
    jne  being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rdx, offset l1
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    int  3 ;force an exception to occur
    ...
l1: ;execution resumes here when exception occurs
    mov  rax, [rcx+8] ;get ContextRecord
    mov  rcx, [rax+48h] ;Dr0
```

```
    or   rcx, [rax+50h] ;Dr1
    or   rcx, [rax+58h] ;Dr2
    or   rcx, [rax+60h] ;Dr3
    jne  being_debugged
```

The values for the debug registers can also be altered prior to resuming execution on the 32-bit versions of *Windows*, which might result in uncontrolled execution unless a software breakpoint is placed at the appropriate location.

B. Instruction Counting

Instruction counting can be performed by registering an exception handler, and then setting hardware breakpoints on particular addresses. When the corresponding address is hit, an EXCEPTION_SINGLE_STEP (0x80000004) exception will be raised. This exception will be passed to the exception handler. The exception handler can choose to adjust the instruction pointer to point to a new instruction, optionally set additional hardware breakpoints on particular addresses, and then resume execution. To set the breakpoints requires access to a context structure. A copy of the context structure can be acquired by calling the kernel32 GetThreadContext() function, which allows setting the initial values for the hardware breakpoints, if necessary. Subsequently, when an exception occurs then the exception handler will receive a copy of the context structure automatically. A debugger will interfere with the single-stepping, resulting in a different count of instructions compared to when a debugger is not present. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l5
    push d fs:[eax]
    mov  fs:[eax], esp
    int  3 ;force exception to occur
l1: nop
```

```
l2: nop
l3: nop
l4: nop
    cmp  al, 4
    jne  being_debugged
    ...
l5: push edi
    mov  eax, [esp+8] ;ExceptionRecord
    mov  edi, [esp+10h] ;ContextRecord
    push 55h ;local-enable DR0, DR1, DR2, DR3
    pop  ecx
    inc  d [ecx*2+edi+0eh] ;Eip
    mov  eax, [eax] ;ExceptionCode
    sub  eax, 80000003h ;EXCEPTION_BREAKPOINT
    jne  l6
    mov  eax, offset l1
    scasd
    stosd ;Dr0
    inc  eax ;l2
    stosd ;Dr1
    inc  eax ;l2
    stosd ;Dr2
    inc  eax ;l4
    stosd ;Dr3
    ;local-enable breakpoints
    ;for compatibility with old CPUs
    mov  ch, 1
    xchg ecx, eax
    scasd
    stosd ;Dr7
    xor  eax, eax
    pop  edi
    ret
l6: dec  eax ;EXCEPTION_SINGLE_STEP
    jne  being_debugged
    inc  b [ecx*2+edi+6] ;Eax
    pop  edi
    ret
```

Since this technique uses a Structured Exception Handler, it cannot be used on the 64-bit versions of *Windows*. The code can be rewritten easily to make use of a Vectored Exception Handler instead. It requires a creating a thread and altering its context, because the debug registers cannot be assigned from inside a vectored exception handler on the 64-bit versions of *Windows*. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows XP* or later:

```
    xor  ebx, ebx
    push eax
    push esp
    push 4 ;CREATE_SUSPENDED
    push ebx
    push offset l1
    push ebx
    push ebx
    call CreateThread
    mov  esi, offset l7
    push esi
    push eax
    xchg ebp, eax
    call GetThreadContext
    mov  eax, offset l2
    lea  edi, [esi+4]
    stosd ;Dr0
    inc  eax
    stosd ;Dr1
    inc  eax
    stosd ;Dr2
    inc  eax
    stosd ;Dr3
    scasd
    push 55h ;local-enable DR0, DR1, DR2, DR3
    pop  eax
    stosd ;Dr7
    push esi
    push ebp
    call SetThreadContext
    push offset l6
    push 1
    call AddVectoredExceptionHandler
    push ebp
    call ResumeThread
    jmp  $
l1: xor  eax, eax
l2: nop
```

```
l3: nop
l4: nop
l5: nop
    cmp  al, 4
    jne  being_debugged
    ...
l6: mov  eax, [esp+4]
    mov  ecx, [eax] ;ExceptionRecord
    ;ExceptionCode
    cmp  [ecx], 80000004h ;EXCEPTION_SINGLE_STEP
    jne  being_debugged
    mov  eax, [eax+4] ;ContextRecord
    cdq
    mov  dh, 1
    inc  b [eax+edx-50h] ;Eax
    inc  d [eax+edx-48h] ;Eip
    or   eax, -1 ;EXCEPTION_CONTINUE_EXECUTION
    ret
l7: dd   10002h ;CONTEXT_i486+CONTEXT_INTEGER
    db   0b0h dup (?)
```

or this 64-bit code to examine the 64-bit *Windows* environment:

```
    push rax
    push rsp
    push 4 ;CREATE_SUSPENDED
    sub  esp, 20h
    xor  r9d, r9d
    mov  r8, offset l1
    xor  edx, edx
    xor  ecx, ecx
    call CreateThread
    mov  ebp, eax
    mov  rsi, offset l7-30h
    push rsi
    pop  rdx
    xchg ecx, eax
    call GetThreadContext
    mov  rax, offset l2
    lea  rdi, [rsi+48h]
    stosq ;Dr0
    inc  rax
    stosq ;Dr1
```

```
    inc  rax
    stosq ;Dr2
    inc  rax
    stosq ;Dr3
    scasq
    push 55h ;local-enable DR0, DR1, DR2, DR3
    pop  rax
    stosd ;Dr7
    push rsi
    pop  rdx
    mov  ecx, ebp
    call SetThreadContext
    mov  rdx, offset l6
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    mov  ecx, ebp
    call ResumeThread
    jmp  $
l1: xor  eax, eax
l2: nop
l3: nop
l4: nop
l5: nop
    cmp  al, 4
    jne  being_debugged
    ...
l6: mov  rax, [rcx] ;ExceptionRecord
    ;ExceptionCode
    cmp  d [rax], 80000004h ;EXCEPTION_SINGLE_STEP
    jne  being_debugged
    mov  rax, [rcx+8] ;ContextRecord
    inc  b [rax+78h] ;Eax
    inc  q [rax+0f8h] ;Eip
    or   eax, -1 ;EXCEPTION_CONTINUE_EXECUTION
    ret
l7: dd   10002h ;CONTEXT_i486+CONTEXT_INTEGER
    db   0c4h dup (?)
```

C. Interrupt 3

Whenever a software interrupt exception occurs, the exception address, and the EIP register value, will point to the instruction after the one that caused the exception. A breakpoint exception is treated as a special case. When an EXCEPTION_BREAKPOINT (0x80000003) exception occurs, *Windows* assumes that it was caused by the one-byte "CC" opcode ("INT 3" instruction). *Windows* decrements the exception address to point to the assumed "CC" opcode, and then passes the exception to the exception handler. The EIP register value is not affected. Thus, if the "CD 03" opcode (long form "INT 03" instruction) is used, the exception address will point to the "03" when the exception handler receives control.

D. Interrupt 0x2d

The interrupt 0x2D is a special case. When it is executed, *Windows* uses the current EIP register value as the exception address, and then it increments by one the EIP register value. However, *Windows* also examines the value in the EAX register to determine how to adjust the exception address. If the EAX register has the value of 1, 3, or 4 on all versions of *Windows*, or the value 5 on *Windows Vista* and later, then *Windows* will increase by one the exception address. Finally, it issues an EXCEPTION_BREAKPOINT (0x80000003) exception if a debugger is present. The interrupt 0x2D behaviour can cause trouble for debuggers. The problem is that some debuggers might use the EIP register value as the address from which to resume, while other debuggers might use the exception address as the address from which to resume. This can result in a single-byte instruction being skipped, or the execution of a completely different instruction because the first byte is missing. These behaviours can be used to infer the presence of the debugger. The check can be made using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
xor  eax, eax ;set Z flag
int  2dh
inc  eax ;debugger might skip
je   being_debugged
```

E. Interrupt 0x41

Interrupt 0x41 can display different behaviour if kernel-mode debugger is present or not. The interrupt 0x41 descriptor normally has a DPL of zero, which means that the interrupt cannot be executed successfully from ring 3. An attempt to execute this interrupt directly will result in a general protection fault (interrupt 0x0D) being issued by the CPU, eventually resulting in an EXCEPTION_ACCESS_VIOLATION (0xC0000005) exception. However, some debuggers hook interrupt 0x41 and adjust its DPL to three, so that the interrupt can be called successfully from user-mode. This fact can be used to infer the presence of a kernel-mode debugger. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    mov  al, 4fh
    int  41h
    jmp  being_debugged
l1: ;execution resumes here if no debugger present
    ...
```

or using this 64-bit code to examine the 64-bit *Windows* environment (though it is unlikely to be supported by a 64-bit debugger):

```
    mov  rdx, offset l1
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    push 4fh
    pop  rax
    int  41h
    jmp  being_debugged
l1: ;execution resumes here if no debugger present
```

...

F.MOV SS

There is a simple trick to detect single-stepping that has worked since the earliest of *Intel* CPUs. It was used quite commonly in the days of DOS, but it still works in all versions of *Windows*. The trick relies on the fact that certain instructions cause all of the interrupts to be disabled while executing the next instruction. In particular, loading the SS register clears interrupts to allow the next instruction to load the [E]SP register without risk of stack corruption. However, there is no requirement that the next instruction loads anything into the [E]SP register. Any instruction can follow the load of the SS register. If a debugger is being used to single-step through the code, then the T flag will be set in the EFLAGS image. This is typically not visible because the T flag will be cleared in the EFLAGS image after each debugger event is delivered. However, if the flags are saved to the stack before the debugger event is delivered, then the T flag will become visible. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push ss
pop  ss
pushfd
test b [esp+1], 1
jne  being_debugged
```

An interesting situation exists in *VirtualPC* when running *Windows 2000*, which is that the CPUID instruction behaves in the same way. It is unknown why this occurs.

There is no 64-bit code example because the SS selector is not supported in that environment.

7.APIs

A debugger can be detected, disabled, or evaded (whereby it loses control on the debuggee), using standard operating system functions. The functions call into several groups, based on common functionality.

A.Heap functions

BasepFreeActivationContextActivationBlock
BasepFreeAppCompatData
ConvertFiberToThread
DeleteFiber
FindVolumeClose
FindVolumeMountPointClose
HeapFree
SortCloseHandle

The one thing that all of these functions have in common is that they call the ntdll RtlFreeHeap() function.

The kernel32 BasepFreeActivationContextActivationBlock() and kernel32 SortCloseHandle() functions exist only on *Windows* 7 and later. The kernel32 BasepFreeAppCompatData() function exists only on *Windows Vista* and later. The kernel32 FindVolumeMountPointClose() calls the ntdll RtlFreeHeap() function only as a special case (specifically, when the hFindVolumeMountPoint parameter is a valid pointer to a handle which can be closed successfully).

However, the point is that the ntdll RtlFreeHeap() function contains a feature that is designed to be used in conjunction with a debugger - a call to the ntdll DbgPrint() function. The problem is that the way in which the ntdll DbgPrint() function is implemented allows an application to detect the presence of a debugger when the function is called.

When the ntdll DbgPrint() function is called, it raises the DBG_PRINTEXCEPTION_C (0x40010006) exception but the exception is handled in a special way, so a registered Structured Exception Handler will not see it. The reason is that *Windows* registers its own Structured Exception Handler internally, which consumes the exception if a debugger does not do so. However, in *Windows XP* and later, any registered Vectored Exception Handler will run before the Structured Exception Handler that *Windows* registers. This might be considered a bug in *Windows*. The presence of a debugger that consumes the exception can now be inferred by the absence of the exception. Further, a different exception is delivered to the Vectored Exception Handler if a debugger is present but did not consume the exception, or if a debugger is not present at all. If a debugger is present but did not consume the exception, then *Windows* will deliver the DBG_PRINTEXCEPTION_C (0x40010006) exception. If a debugger is not present, then Windows will deliver the EXCEPTION_ACCESS_VIOLATION (0xC0000005) exception. The presence of a debugger can now be inferred by either the absence of the exception, or the value of the exception.

There is an additional case, which applies to heap and resource functions, among others, whereby the functions can be forced to cause a debug break. What they have in common is a check of the BeingDebugged flag in the Process Environment Block. The presence of the debugger can be faked, to force the interrupt 3 exception to occur, and the exception should be visible to the debuggee. Thus, if the exception is missing (because the debugger consumed it), then the debugger's presence is revealed. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit for 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
```

```
    mov  fs:[eax], esp
    ;Process Environment Block
    mov  eax, fs:[eax+30h]
    inc  b [eax+2] ;set BeingDebugged
    push offset l2
    call HeapDestroy
    jmp  being_debugged
l1: ;execution resumes here due to exception
    ...
l2: db   0ch dup (0)
    dd   40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
    db   30h dup (0)
    dd   40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
    db   24h dup (0)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rdx, offset l1
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    push 60h
    pop  rsi
    gs:lodsq ;Process Environment Block
    inc  b [rax+2] ;set BeingDebugged
    mov  rcx, offset l2
    call HeapDestroy
    jmp  being_debugged
l1: ;execution resumes here due to exception
    ...
l2: db   14h dup (0)
    dd   40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
    db   58h dup (0)
    dd   40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
    db   30h dup (0)
```

The flag value appears twice in each case, because it is placed in both possible locations for the flag field, depending on the version of *Windows.* This avoids the need for a version check.

Note that on *Windows Vista* and later, the behaviour changed slightly. Previously, the debug break was caused by a call to the ntdll DbgBreakPoint() function. Now, it is caused by an interrupt 3 instruction that is stored directly into the code stream. The outcome is the same in either case, though.

The detection can also be extended slightly. The LastErrorValue in the Thread Environment Block can be set to zero prior to calling the function, either directly, or by calling the kernel32 SetLastError() function,. If no exception occurred, then on return from the function, the value in that field (also returned by the kernel32 GetLastError() function) will be set to ERROR_INVALID_HANDLE (6).

B.Handles

- OpenProcess
- CloseHandle
- CreateFile
- LoadLibrary
- ReadFile

i.OpenProcess

The kernel32 OpenProcess() function (or the ntdll NtOpenProcess() function) has at times been claimed to detect the presence of a debugger when used on the "csrss.exe" process. This is incorrect. While it is true that the function call will succeed in the presence of some debuggers, this is due to a side-effect of the debugger's behaviour (specifically, acquiring the debug privilege), and not due to the debugger itself (this should be obvious since the function call does not succeed when used with certain debuggers). All it reveals is that the user account for the process is a member of the administrators group and it has the debug privilege. The reason is that the success or failure of the function call is limited only by the process privilege level. If the user account of the process is a member of the administrators group and has the debug privilege, then the function call will succeed; if not, then not. It is not sufficient for a standard user to acquire the debug privilege, nor can an administrator call the function successfully without it. The process ID of the csrss.exe process can be acquired by the ntdll CsrGetProcessId() function on *Windows XP* and later (other methods exist for earlier versions of *Windows*, and are shown later in the context of finding the "Explorer.exe" process). The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    call CsrGetProcessId
    push eax
    push 0
    push 1f0fffh ;PROCESS_ALL_ACCESS
    call OpenProcess
    test eax, eax
    jne  admin_with_debug_priv
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    call CsrGetProcessId
    push rax
    pop  r8
    cdq
    mov  ecx, 1f0fffh ;PROCESS_ALL_ACCESS
    call OpenProcess
    test eax, eax
    jne  admin_with_debug_priv
```

The debug privilege can be acquired using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  ebx, ebx
    push 2 ;SE_PRIVILEGE_ENABLED
    push ebx
    push ebx
    push esp
    push offset l1
    push ebx
```

```
    call LookupPrivilegeValueA
    push eax
    push esp
    push 20h ;TOKEN_ADJUST_PRIVILEGES
    push -1 ;GetCurrentProcess()
    call OpenProcessToken
    pop  ecx
    push eax
    mov  eax, esp
    push ebx
    push ebx
    push ebx
    push eax
    push ebx
    push ecx
    call AdjustTokenPrivileges
    ...
l1: db   "SeDebugPrivilege", 0
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    xor  ebx, ebx
    push 2 ;SE_PRIVILEGE_ENABLED
    push rbx
    push rbx
    mov  r8d, esp
    mov  rdx, offset l1
    xor  ecx, ecx
    call LookupPrivilegeValueA
    push rax
    mov  r8d, esp
    push 20h ;TOKEN_ADJUST_PRIVILEGES
    pop  rdx
    or   rcx, -1 ;GetCurrentProcess()
    call OpenProcessToken
    pop  rcx
    push rax
    mov  r8d, esp
    push rbx
    push rbx
    sub  esp, 20h
    xor  r9d, r9d
    cdq
```

```
    call AdjustTokenPrivileges
    ...
l1: db   "SeDebugPrivilege", 0
```

ii.CloseHandle

One well-known technique for detecting a debugger involves the kernel32 CloseHandle() function. If an invalid handle is passed to the kernel32 CloseHandle() function (or directly to the ntdll NtClose() function, or the kernel32 FindVolumeMountPointClose() function on *Windows 2000* and later (which simply calls the kernel32 CloseHandle() function)), and a debugger is present, then an EXCEPTION_INVALID_HANDLE (0xC0000008) exception will be raised. This exception can be intercepted by an exception handler, and is an indication that a debugger is running. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset being_debugged
    push d fs:[eax]
    mov  fs:[eax], esp
    ;any illegal value will do
    ;must be dword-aligned
    ;on Windows Vista and later
    push esp
    call CloseHandle
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rdx, offset being_debugged
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    ;any illegal value will do
    ;must be dword-aligned
    ;on Windows Vista and later
    mov  ecx, esp
```

```
    call CloseHandle
```

However, there is a second case which involves using a protected handle instead. If a protected handle is passed to the kernel32 CloseHandle() function (or directly to the ntdll NtClose() function), and a debugger is present, then an EXCEPTION_HANDLE_NOT_CLOSABLE (0xC0000235) exception will be raised. This exception can be intercepted by an exception handler, and is an indication that a debugger is running. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset being_debugged
    push d fs:[eax]
    mov  fs:[eax], esp
    push eax
    push eax
    push 3 ;OPEN_EXISTING
    push eax
    push eax
    push eax
    push offset l1
    call CreateFileA
    push 2 ;HANDLE_FLAG_PROTECT_FROM_CLOSE
    push -1
    push eax
    xchg ebx, eax
    call SetHandleInformation
    push ebx
    call CloseHandle
    ...
l1: db   "myfile", 0
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rdx, offset being_debugged
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
```

```
    cdq
    push rdx
    push rdx
    push 3 ;OPEN_EXISTING
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    mov  rcx, offset l1
    call CreateFileA
    mov  ebx, eax
    push 2 ;HANDLE_FLAG_PROTECT_FROM_CLOSE
    pop  r8
    or   rdx, -1
    xchg ecx, eax
    call SetHandleInformation
    mov  ecx, ebx
    call CloseHandle
    ...
l1: db   "myfile", 0
```

To (attempt to) defeat any of these methods is easiest on *Windows XP* and later, where a FirstHandler Vectored Exception Handler can be registered by the debugger to hide the exception and silently resume execution. However, there is the problem of transparently hooking the kernel32 AddVectoredExceptionHandler() function (or the ntdll RtlAddVectoredExceptionHandler() function), in order to prevent another handler from registering as the first handler. It is not enough to intercept an attempt to register a first handler and then make it the last handler. The reason is that it would be revealed by registering two handlers and then causing an exception, because the handlers would be called in the wrong order. There is also the potential problem of something hooking the function and detecting such a changed request, and then simply changing it back again. Another way to fix it might be to disassemble the function to find the base pointer that holds the list head, but this is platform-specific. Of course, since the function returns a pointer to the handler structure, the list can be traversed by registering a handler and then parsing the structure.

This situation is still better than the problem of transparently hooking the ntdll NtClose() function on *Windows NT* and *Windows 2000*, in order to register a Structured Exception Handler to hide the exception.

There is a flag that can be set to produce the exceptional behaviour, even if no debugger is present. By setting the FLG_ENABLE_CLOSE_EXCEPTIONS (0x400000) flag in the "HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag" registry value, and then rebooting, the kernel32 CloseHandle() function, and the ntdll NtClose() function, will always raise an exception if an invalid or protected handle is passed to the function. The effect is system-wide, and is supported on all *Windows NT*-based versions of *Windows*, both 32-bit and 64-bit.

There is another flag that results in similar behaviour for other functions that accept handles. By setting the FLG_APPLICATION_VERIFIER (0x100) flag in the "HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag" registry value, and then rebooting, the ntoskrnl ObReferenceObjectByHandle() function will always raise an exception if an invalid handle is passed to a function (such as the kernel32 SetEvent() function) that calls the ntoskrnl ObReferenceObjectByHandle() function. The effect is also system-wide.

Note that one of these flags is currently documented incorrectly, and the other of these flags is currently documented incompletely. The "Enable close exception" flag[1] is documented as causing exceptions to be raised for invalid handles that are passed to functions other than the ntoskrnl NtClose() function, but this is incorrect; the

[1] http://msdn.microsoft.com/en-us/library/ff542887.aspx

"Enable bad handles detection" flag[2] will cause exceptions to be raised for invalid handles that are passed to functions other than the ntoskrnl NtClose() function, but this behaviour is not documented.

iii.CreateFile

A slightly unreliable way to detect the presence of a debugger is to attempt to open exclusively the file of current process. When some debuggers are present, this action will always fail. The reason is that when a process is started for debugging, a handle to the file is opened. This allows the debugger to read the debug information from the file (assuming that it is present). The handle value is stored in the structure that is filled when the CREATE_PROCESS_DEBUG_EVENT event occurs. If that handle is not closed by the debugger, then the file cannot be opened for exclusive access. Since the debugger did not open the file, it would be easy to forget to close it.

Of course, if any other application (such as a hex editor) is examining the file, then the open will also fail for the same reason. This is why the technique is considered to be unreliable, but depending on the intention, such false positives it might be acceptable. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push 104h ;MAX_PATH
mov  ebx, offset l1
push ebx
push 0 ;self filename
call GetModuleFileNameA
cdq
push edx
```

[2] http://msdn.microsoft.com/en-us/library/ff542881.aspx

```
    push edx
    push 3 ;OPEN_EXISTING
    push edx
    push edx
    inc  edx
    ror  edx, 1
    push edx ;GENERIC_READ
    push ebx
    call CreateFileA
    inc  eax
    je   being_debugged
    ...
l1: db   104h dup (?) ;MAX_PATH
```

or using this 64-bit code to examine the 64-bit *Windows* environment (but the technique does not work for 64-bit processes):

```
    mov  r8d, 104h ;MAX_PATH
    mov  rbx, offset l1
    push rbx
    pop  rdx
    xor  ecx, ecx ;self filename
    call GetModuleFileNameA
    cdq
    push rdx
    push rdx
    push 3 ;OPEN_EXISTING
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    inc  edx
    ror  edx, 1 ;GENERIC_READ
    push rbx
    pop  rcx
    call CreateFileA
    inc  eax
    je   being_debugged
    ...
l1: db   104h dup (?) ;MAX_PATH
```

The kernel32 CreateFile() function can also be used to detect the presence of kernel-mode drivers which might belong to a debugger (or any other tool of interest). Tools that make use of kernel-mode drivers also need a way to communicate with those drivers. A very common method is through the use of named devices. Thus, by attempting to open such a device, any success indicates the presence of the driver. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor   eax, eax
    mov   edi, offset l2
l1: push  eax
    push  eax
    push  3 ;OPEN_EXISTING
    push  eax
    push  eax
    push  eax
    push  edi
    call  CreateFileA
    inc   eax
    jne   being_debugged
    or    ecx, -1
    repne scasb
    cmp   [edi], al
    jne   l1
    ...
l2: <array of ASCIIZ strings, zero to end>
```

A typical list includes the following names:

```
    db    "\\.\EXTREM", 0 ;Phant0m
    db    "\\.\FILEM", 0 ;FileMon
    db    "\\.\FILEVXG", 0 ;FileMon
    db    "\\.\ICEEXT", 0 ;SoftICE Extender
    db    "\\.\NDBGMSG.VXD", 0 ;SoftICE
    db    "\\.\NTICE", 0 ;SoftICE
    db    "\\.\REGSYS", 0 ;RegMon
    db    "\\.\REGVXG", 0 ;RegMon
    db    "\\.\RING0", 0 ;Olly Advanced
    db    "\\.\SICE", 0 ;SoftICE
    db    "\\.\SIWVID", 0 ;SoftICE
    db    "\\.\TRW", 0 ;TRW
    db    "\\.\SPCOMMAND", 0 ;Syser
    db    "\\.\SYSER", 0 ;Syser
```

```
    db    "\\.\SYSERBOOT", 0 ;Syser
    db    "\\.\SYSERDBGMSG", 0 ;Syser
    db    "\\.\SYSERLANGUAGE", 0 ;Syser
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov   rdi, offset l2
l1: xor   edx, edx
    push  rdx
    push  rdx
    push  3 ;OPEN_EXISTING
    sub   esp, 20h
    xor   r9d, r9d
    xor   r8d, r8d
    push  rdi
    pop   rcx
    call  CreateFileA
    inc   eax
    jne   being_debugged
    or    ecx, -1
    repne scasb
    cmp   [rdi], al
    jne   l1
    ...
l2: <array of ASCIIZ strings, zero to end>
```

However, there is currently no 64-bit list because of a shortage of kernel-mode debuggers that offer user-mode services.

Note that the "\\.\NTICE" driver name is valid only for *SoftICE* prior to version 4.0. *SoftICE* v4.x does not create a device with such a name in *Windows NT*-based platforms. Instead, the device name is "\\.\NTICExxxx"), where "xxxx" is four hexadecimal characters. The source of the characters is the 9th, the 7th, the 5th, and the 3rd, character from the data in the "Serial" registry value. This value appears in multiple places in the registry. The *SoftICE* driver uses the "HKLM\System\CurrentContrlSet\Services\NTice\Serial" registry value. The nmtrans DevIO_ConnectToSoftICE() function uses the

"HKLM\Software\NuMega\SoftIce\Serial" registry value. The algorithm that SoftICE uses is to reverse the string, then, beginning with the third character, take every second character, for four characters. There is a simpler method to achieve this, of course. The name can be constructed using this 32-bit code to examine the 32-bit *Windows* environment on the 32-bit versions of *Windows* (*SoftICE* does not run on the 64-bit versions of *Windows*):

```
    xor  ebx, ebx
    push eax
    push esp
    push 1 ;KEY_QUERY_VALUE
    push ebx
    push offset l2
    push 80000002h ;HKLM
    call RegOpenKeyExA
    pop  ecx
    push 0dh ;sizeof(l3)
    push esp
    mov  esi, offset l3
    push esi
    push eax ;REG_NONE
    push eax
    push offset l4
    push ecx
    call RegQueryValueExA
    push 4
    pop  ecx
    mov  edi, offset l6
l1: mov  al, [ecx*2+esi+1]
    stosb
    loop l1
    push ebx
    push ebx
    push 3 ;OPEN_EXISTING
    push ebx
    push ebx
    push ebx
    push offset l5
    call CreateFileA
    inc  eax
```

```
    jne  being_debugged
    ...
l2: db   "Software\NuMega\SoftIce", 0
l3: db   0dh dup (?)
l4: db   "Serial", 0
l5: db   "\\.\ntice"
l6: db   "xxxx", 0
```

iv.LoadLibrary

The kernel32 LoadLibrary() function is a surprisingly simple and effective way to detect a debugger. When a file is loaded in the presence of a debugger, using the kernel32 LoadLibrary() function (or any of its variations – the kernel32 LoadLibraryEx() function, or the ntdll LdrLoadDll() function), a handle to the file is opened. This allows the debugger to read the debug information from the file (assuming that it is present). The handle value is stored in the structure that is filled when the LOAD_DLL_DEBUG_EVENT event occurs. If that handle is not closed by the debugger (unloading the DLL will not close it), then the file cannot be opened for exclusive access. Since the debugger did not open the file, it would be easy to forget to close it. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  ebx, offset l1
    push ebx
    call LoadLibraryA
    cdq
    push edx
    push edx
    push 3 ;OPEN_EXISTING
    push edx
    push edx
    inc  edx
    ror  edx, 1 ;GENERIC_READ
    push edx
    push ebx
```

```
    call CreateFileA
    inc  eax
    je   being_debugged
    ...
l1: db   "myfile.", 0
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rbx, offset l1
    push rbx
    pop  rcx
    call LoadLibraryA
    xor  edx, edx
    push rdx
    push rdx
    push 3 ;OPEN_EXISTING
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    inc  edx
    ror  edx, 1 ;GENERIC_READ
    push rbx
    pop  rcx
    call CreateFileA
    inc  eax
    je   being_debugged
    ...
l1: db   "myfile.", 0
```

An alternative method is to call another function that calls the kernel32 CreateFile() function (or any of its variations - the ntdll NtCreateFile() function, or the ntdll NtOpenFile() function) internally. One example is the resource-updating functions, such as the kernel32 EndUpdateResource() function. The reason why the kernel32 EndUpdateResource() function works is because it eventually calls the kernel32 CreateFile() function to write the new resource table. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  ebx, offset l1
    push ebx
    call LoadLibraryA
    push 0
    push ebx
    call BeginUpdateResourceA
    push 0
    push eax
    call EndUpdateResourceA
    test eax, eax
    je   being_debugged
    ...
l1: db "myfile.", 0
```

or using this 64-bit code to examine the 64-bit *Windows* environment (but the technique does not work for 64-bit processes):

```
    mov  rbx, offset l1
    push rbx
    pop  rcx
    call LoadLibraryA
    xor  edx, edx
    push rbx
    pop  rcx
    call BeginUpdateResourceA
    cdq
    xchg ecx, eax
    call EndUpdateResourceA
    test eax, eax
    je   being_debugged
    ...
l1: db "myfile.", 0
```

v.ReadFile

The kernel32 ReadFile() function can be used to perform self-modification of the code stream, by reading file content to a location after the call to the function. It can also be used to remove software breakpoints that a debugger might place in the code stream, particularly right after the call to the function. The result in that case would be that the code executes freely. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push 104h ;MAX_PATH
    mov  ebx, offset l2
    push ebx
    push 0 ;self filename
    call GetModuleFileNameA
    cdq
    push edx
    push edx
    push 3 ;OPEN_EXISTING
    push edx
    ;FILE_SHARE_READ
    ;because a debugger might prevent
    ;exclusive access to the running file
    inc  edx
    push edx
    ror  edx, 1
    push edx ;GENERIC_READ
    push ebx
    call CreateFileA
    push 0
    push esp
    push 1 ;more bytes might be more useful
    push offset l1
    push eax
    call ReadFile
l1: int  3 ;replaced by "M" from the MZ header
    ...
l2: db   104h dup (?) ;MAX_PATH
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  r8d, 104h ;MAX_PATH
    mov  rbx, offset l2
    push rbx
    pop  rdx
    xor  ecx, ecx ;self filename
    call GetModuleFileNameA
    cdq
    push rdx
```

```
    push rdx
    push 3 ;OPEN_EXISTING
    sub  esp, 20h
    xor  r9d, r9d
    ;FILE_SHARE_READ
    ;because a debugger might prevent
    ;exclusive access to the running file
    inc  edx
    push rdx
    pop  r8
    ror  edx, 1 ;GENERIC_READ
    push rbx
    pop  rcx
    call CreateFileA
    push 0
    mov  r9d, esp
    sub  esp, 20h
    push 1 ;more bytes might be more useful
    pop  r8
    mov  rdx, offset l1
    xchg ecx, eax
    call ReadFile
l1: int  3 ;replaced by "M" from the MZ header
    ...
l2: db   104h dup (?) ;MAX_PATH
```

One way to defeat this technique is to use hardware breakpoints instead of software breakpoints when stepping over function calls.

C.Execution Timing

```
    RDPMC
    RDTSC
    GetLocalTime
    GetSystemTime
    GetTickCount
    KiGetTickCount
    QueryPerformanceCounter
    timeGetTime
```

When a debugger is present, and used to single-step through the code, there is a significant delay between the executions of the individual instructions, when compared to native execution. This delay can be measured using one of several possible time sources. These sources include the RDPMC instruction (however, this instruction requires that the PCE flag is set in the CR4 register, but this is not the default setting), the RDTSC instruction (however, this instruction requires that the TSD flag is clear in the CR4 register, but this is the default setting), the kernel32 GetLocalTime() function, the kernel32 GetSystemTime() function, the kernel32 QueryPerformanceCounter() function, the kernel32 GetTickCount() function, the ntoskrnl KiGetTickCount() function (exposed via the interrupt 0x2A interface on the 32-bit versions of *Windows*), and the winmm timeGetTime() function. However, the resolution of the winmm timeGetTime() function is variable, depending on whether or not it branches internally to the kernel32 GetTickCount() function, making it very unreliable to measure small intervals. The RDMSR instruction can also be used as a time source, but it cannot be used in user-mode. The check can be made for the RDPMC instruction using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
    xor  ecx, ecx ;read 32-bit counter 0
    rdpmc
    xchg ebx, eax
    rdpmc
    sub  eax, ebx
    cmp  eax, 500h
    jnbe being_debugged
```

The check can be made for the RDTSC instruction using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
    rdtsc
    xchg esi, eax
    mov  edi, edx
```

```
    rdtsc
    sub  eax, esi
    sbb  edx, edi
    jne  being_debugged
    cmp  eax, 500h
    jnbe being_debugged
```

The check can be made for the kernel32 GetLocalTime() function using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  ebx, offset l1
    push ebx
    call GetLocalTime
    mov  ebp, offset l2
    push ebp
    call GetLocalTime
    mov  esi, offset l3
    push esi
    push ebx
    call SystemTimeToFileTime
    mov  edi, offset l4
    push edi
    push ebp
    call SystemTimeToFileTime
    mov  eax, [edi]
    sub  eax, [esi]
    mov  edx, [edi+4]
    sbb  edx, [esi+4]
    jne  being_debugged
    cmp  eax, 10h
    jnbe being_debugged
    ...
l1: db   10h dup (?) ;sizeof(SYSTEMTIME)
l2: db   10h dup (?) ;sizeof(SYSTEMTIME)
l3: db   8 dup (?) ;sizeof(FILETIME)
l4: db   8 dup (?) ;sizeof(FILETIME)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rbx, offset l1
    push rbx
```

```
    pop  rcx
    call GetLocalTime
    mov  rbp, offset l2
    push rbp
    pop  rcx
    call GetLocalTime
    mov  rsi, offset l3
    push rsi
    pop  rdx
    push rbx
    pop  rcx
    call SystemTimeToFileTime
    mov  rdi, offset l4
    push rdi
    pop  rdx
    push rbp
    pop  rcx
    call SystemTimeToFileTime
    mov  rax, [rdi]
    sub  rax, [rsi]
    cmp  rax, 10h
    jnbe being_debugged
    ...
l1: db   10h dup (?) ;sizeof(SYSTEMTIME)
l2: db   10h dup (?) ;sizeof(SYSTEMTIME)
l3: db   8 dup (?) ;sizeof(FILETIME)
l4: db   8 dup (?) ;sizeof(FILETIME)
```

The check can be made for the kernel32 GetSystemTime() function using exactly the same code as for the kernel32 GetLocalTime() function, apart from changing the function name.

The check can be made for the kernel32 GetTickCount() function using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
    call GetTickCount
    xchg ebx, eax
    call GetTickCount
    sub  eax, ebx
    cmp  eax, 10h
    jnbe being_debugged
```

The check can be made for the ntoskrnl KiGetTickCount() function using this 32-bit code to examine the 32-bit *Windows* environment on the 32-bit versions of *Windows* (the interrupt is not supported on the 64-bit versions of *Windows*):

```
    int  2ah
    xchg ebx, eax
    int  2ah
    sub  eax, ebx
    cmp  eax, 10h
    jnbe being_debugged
```

The check can be made for the kernel32 QueryPerformanceCounter() function using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  esi, offset l1
    push esi
    call QueryPerformanceCounter
    mov  edi, offset l2
    push edi
    call QueryPerformanceCounter
    mov  eax, [edi]
    sub  eax, [esi]
    mov  edx, [edi+4]
    sbb  edx, [esi+4]
    jne  being_debugged
    cmp  eax, 10h
    jnbe being_debugged
    ...
l1: db   8 dup (?) ;sizeof(LARGE_INTEGER)
l2: db   8 dup (?) ;sizeof(LARGE_INTEGER)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rsi, offset l1
    push rsi
    pop  rcx
    call QueryPerformanceCounter
    mov  rdi, offset l2
```

```
    push rdi
    pop  rcx
    call QueryPerformanceCounter
    mov  rax, [rdi]
    sub  rax, [rsi]
    cmp  rax, 10h
    jnbe being_debugged
    ...
l1: db   8 dup (?) ;sizeof(LARGE_INTEGER)
l2: db   8 dup (?) ;sizeof(LARGE_INTEGER)
```

The check can be made for the winmm timeGetTime() function using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
    call timeGetTime
    xchg ebx, eax
    call timeGetTime
    sub  eax, ebx
    cmp  eax, 10h
    jnbe being_debugged
```

D.Process-level

CheckRemoteDebuggerPresent
CreateToolhelp32Snapshot
DbgSetDebugFilterState
IsDebuggerPresent
NtQueryInformationProcess
RtlQueryProcessHeapInformation
RtlQueryProcessDebugInformation
SwitchToThread
Toolhelp32ReadProcessMemory
UnhandledExceptionFilter

i.CheckRemoteDebuggerPresent

The kernel32 CheckRemoteDebuggerPresent() function was introduced in *Windows XP* SP1, to query a value that has existed since *Windows NT*. "Remote" in this sense refers to a separate process on the same machine. The function sets to 0xffffffff the value to which the pbDebuggerPresent argument points, if a debugger is present (that is, attached to the current process). The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
push esp
push -1 ;GetCurrentProcess()
call CheckRemoteDebuggerPresent
pop  eax
test eax, eax
jne  being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
enter 20h, 0
mov   edx, ebp
or    rcx, -1 ;GetCurrentProcess()
call  CheckRemoteDebuggerPresent
leave
test  ebp, ebp
jne   being_debugged
```

ii.Parent Process

Users typically execute applications by clicking on an icon which is displayed by the shell process (Explorer.exe). As a result, the parent process of the executed process will be Explorer.exe. Of course, if the application is executed from the command-line, then the parent process of the executed process will be the command window process. Executing an application by debugging it will cause the parent process of the executed process to be the debugger process.

Executing applications from the command-line can cause problems for certain applications, because they expect the parent process to be Explorer.exe.

Some applications check the parent process name, expecting it to be "Explorer.exe". Some applications compare the parent process ID against that of Explorer.exe. A mismatch in either case might result in the application thinking that it is being debugged.

At this point, we take a slight detour, and introduce a topic that should logically come later. The simplest way to obtain the process ID of Explorer.exe is by calling the user32 GetShellWindow() and user32 GetWindowThreadProcessId() functions. That leaves the process ID and name of the parent process of the current process, which can be obtained by calling the ntdll NtQueryInformationProcess() function with the ProcessBasicInformation class. The calls can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    call GetShellWindow
    push eax
    push esp
    push eax
    call GetWindowThreadProcessId
    push 0
    push 18h ;sizeof(PROCESS_BASIC_INFORMATION)
    mov  ebp, offset l1
    push ebp
    push 0 ;ProcessBasicInformation
    push -1 ;GetCurrentProcess()
    call NtQueryInformationProcess
    pop  eax
    ;InheritedFromUniqueProcessId
    cmp  [ebp+14h], eax
    jne  being_debugged
    ...
    ;sizeof(PROCESS_BASIC_INFORMATION)
l1: db   18h dup (?)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
     call  GetShellWindow
     enter 20h, 0
     mov   edx, ebp
     xchg  ecx, eax
     call  GetWindowThreadProcessId
     leave
     push  0
     sub   esp, 20h
     push  30h ;sizeof(PROCESS_BASIC_INFORMATION)
     pop   r9
     mov   rbx, offset l1
     push  rbx
     pop   r8
     cdq ;ProcessBasicInformation
     or    rcx, -1 ;GetCurrentProcess()
     call  NtQueryInformationProcess
     ;InheritedFromUniqueProcessId
     cmp   [rbx+20h], ebp
     jne   being_debugged
     ...
     ;sizeof(PROCESS_BASIC_INFORMATION)
l1:  db   30h dup (?)
```

However, this code has a serious problem, which is that there can be multiple instances of Explorer.exe within a single session, if the "HKCU \Software\Microsoft\Windows\CurrentVersion\Explorer\ Advanced\SeparateProcess" registry value (introduced in *Windows 2000*) is non-zero. This has the effect of running a separate copy of Explorer.exe for every window that is opened. As a result, the shell window might not be the parent process of the current process, and yet Explorer.exe is the parent process name.

iii.CreateToolhelp32Snapshot

The process ID of both Explorer.exe and the parent process of the current process, and the name of that parent process, can be obtained by the kernel32 CreateToolhelp32Snapshot() function and a kernel32 Process32Next() function enumeration. The call can be made using this 32-bit code to examine the 32-bit

Windows environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor   esi, esi
    xor   edi, edi
    push  esi
    push  2 ;TH32CS_SNAPPROCESS
    call  CreateToolhelp32Snapshot
    mov   ebx, offset l9
    xchg  ebp, eax
l1: push  ebx
    push  ebp
    call  Process32First
l2: mov   eax, fs:[eax+1fh] ;UniqueProcess
    cmp   [ebx+8], eax ;th32ProcessID
    cmove edi, [ebx+18h] ;th32ParentProcessID
    test  edi, edi
    je    l3
    cmp   esi, edi
    je    l7
l3: lea   ecx, [ebx+24h] ;szExeFile
    push  esi
    mov   esi, ecx
l4: lodsb
    cmp   al, "\"
    cmove ecx, esi
    or    b [esi-1], " "
    test  al, al
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    push  edi
    mov   edi, offset l8
    repe  cmpsb
    pop   edi
    pop   esi
    jne   l6
    test  esi, esi
    je    l5
    mov   esi, offset l10
    cmp   cl, [esi]
    adc   [esi], ecx
l5: mov   esi, [ebx+8] ;th32ProcessID
l6: push  ebx
```

```
    push  ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    dec   b [offset l10+1]
    jne   l1
    jmp   being_debugged
l7: ...
    ;trailing zero is converted to space
l8: db    "explorer.exe "
l9: dd    128h ;sizeof(PROCESSENTRY32)
    db    124h dup (?)
l10:db    0ffh, 1, ?, ?
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   esi, esi
    xor   edi, edi
    xor   edx, edx
    push  2 ;TH32CS_SNAPPROCESS
    pop   rcx
    call  CreateToolhelp32Snapshot
    mov   rbx, offset l9
    xchg  ebp, eax
l1: push  rbx
    pop   rdx
    mov   ecx, ebp
    call  Process32First
l2: mov   eax, gs:[rax+3fh] ;UniqueProcess
    cmp   [rbx+8], eax ;th32ProcessID
    cmove edi, [rbx+20h] ;th32ParentProcessID
    test  esi, esi
    je    l3
    cmp   esi, edi
    je    l7
l3: lea   ecx, [rbx+2ch] ;szExeFile
    push  rsi
    mov   esi, ecx
l4: lodsb
    cmp   al, "\"
    cmove ecx, esi
    or    b [rsi-1], " "
    test  al, al
```

```
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    push  rdi
    mov   rdi, offset l8
    repe  cmpsb
    pop   rdi
    pop   rsi
    jne   l6
    test  esi, esi
    je    l5
    mov   rsi, offset l10
    cmp   cl, [rsi]
    adc   [rsi], ecx
l5: mov   esi, [rbx+8] ;th32ProcessID
l6: push  rbx
    pop   rdx
    mov   ecx, ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    dec   b [offset l10+1]
    jne   l1
    jmp   being_debugged
l7: ...
    ;trailing zero is converted to space
l8: db    "explorer.exe "
l9: dd    130h ;sizeof(PROCESSENTRY32)
    db    12ch dup (?)
l10:db    0ffh, 1, ?, ?
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger. A common technique that attempts to defeat it is to force the kernel32 Process32Next() function to return FALSE, which causes the loop to exit early. However, it should be a suspicious condition if either Explorer.exe or the current process was not seen.

The code will run in a single pass if only one copy of Explorer.exe exists. If there are multiple copies, then the code will perform a second pass.

On the first pass, the parent process ID will be obtained. On the second pass, the parent process ID will be compared against the process ID of each instance of Explorer.exe.

There is a minor problem with this code, though, which is that if multiple users are logged on at the same time, then their processes will be visible, too. At least one of them will also be Explorer.exe, and it will not be the parent process of any process in this session. This will cause the code to run two passes, even if only one of them would be sufficient because only one instance of Explorer.exe exists in the current session.

One way to avoid that problem is to determine the user-name and domain name of the process, and match that before accepting the process as found. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor   esi, esi
    xor   edi, edi
    push  esi
    push  2 ;TH32CS_SNAPPROCESS
    call  CreateToolhelp32Snapshot
    mov   ebx, offset l16
    xchg  ebp, eax
l1: push  ebx
    push  ebp
    call  Process32First
l2: mov   eax, fs:[eax+1fh] ;UniqueProcess
    cmp   [ebx+8], eax ;th32ProcessID
    cmove edi, [ebx+18h] ;th32ParentProcessID
    test  edi, edi
    je    l3
    cmp   esi, edi
    je    l9
l3: lea   ecx, [ebx+24h] ;szExeFile
    push  esi
    mov   esi, ecx
l4: lodsb
    cmp   al, "\"
```

```
    cmove ecx, esi
    or    b [esi-1], " "
    test  al, al
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    push  edi
    mov   edi, offset l15
    repe  cmpsb
    pop   edi
    pop   esi
    jne   l8
    mov   eax, [ebx+8] ;th32ProcessID
    push  ebx
    push  ebp
    push  esi
    push  edi
    call  l10
    dec   ecx ;invert Z flag
    jne   l6
    push  ebx
    push  edi
    dec   ecx
    call  l11
    pop   esi
    pop   edx
    mov   cl, 2
    ;compare user names
    ;then domain names
l5: lodsb
    scasb
    jne   l6
    test  al, al
    jne   l5
    mov   esi, ebx
    mov   edi, edx
    loop  l5
l6: pop   edi
    pop   esi
    pop   ebp
    pop   ebx
    jne   l8
    test  esi, esi
    je    l7
```

```
    mov   esi, offset l17
    cmp   cl, [esi]
    adc   [esi], ecx
l7: mov   esi, [ebx+8] ;th32ProcessID
l8: push  ebx
    push  ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    dec   b [offset l17+1]
    jne   l1
    jmp   being_debugged
l9: ...
l10:push  eax
    push  0
    push  400h ;PROCESS_QUERY_INFORMATION
    call  OpenProcess
    xchg  ecx, eax
    jecxz l14
l11:push  eax
    push  esp
    push  8 ;TOKEN_QUERY
    push  ecx
    call  OpenProcessToken
    pop   ebx
    xor   ebp, ebp
l12:push  ebp
    push  0 ;GMEM_FIXED
    call  GlobalAlloc
    push  eax
    push  esp
    push  ebp
    push  eax
    push  1 ;TokenUser
    push  ebx
    xchg  esi, eax
    call  GetTokenInformation
    pop   ebp
    xchg  ecx, eax
    jecxz l12
    xor   ebp, ebp
l13:push  ebp
    push  0 ;GMEM_FIXED
    call  GlobalAlloc
```

```
    xchg  ebx, eax
    push  ebp
    push  0 ;GMEM_FIXED
    call  GlobalAlloc
    xchg  edi, eax
    push  eax
    mov   eax, esp
    push  ebp
    mov   ecx, esp
    push  ebp
    mov   edx, esp
    push  eax
    push  ecx
    push  ebx
    push  edx
    push  edi
    push  d [esi]
    push  0
    call  LookupAccountSidA
    pop   ecx
    pop   ebp
    pop   edx
    xchg  ecx, eax
    jecxz l13
l14:ret
    ;trailing zero is converted to space
l15:db    "explorer.exe "
l16:dd    128h ;sizeof(PROCESSENTRY32)
    db    124h dup (?)
l17:db    0ffh, 1, ?, ?
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   esi, esi
    xor   edi, edi
    xor   edx, edx
    push  2 ;TH32CS_SNAPPROCESS
    pop   rcx
    call  CreateToolhelp32Snapshot
    mov   rbx, offset l16
    xchg  ebp, eax
l1: push  rbx
    pop   rdx
```

```
    mov   ecx, ebp
    call  Process32First
l2: mov   eax, gs:[rax+3fh] ;UniqueProcess
    cmp   [rbx+8], eax ;th32ProcessID
    cmove edi, [rbx+20h] ;th32ParentProcessID
    test  esi, esi
    je    l3
    cmp   esi, edi
    je    l9
l3: lea   ecx, [rbx+2ch] ;szExeFile
    push  rsi
    mov   esi, ecx
l4: lodsb
    cmp   al, "\"
    cmove ecx, esi
    or    byte [rsi-1], " "
    test  al, al
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    push  rdi
    mov   rdi, offset l15
    repe  cmpsb
    pop   rdi
    pop   rsi
    jne   l8
    mov   r8d, [rbx+8] ;th32ProcessID
    push  rbx
    push  rbp
    push  rsi
    push  rdi
    call  l10
    dec   ecx ;invert Z flag
    jne   l6
    push  rbx
    push  rdi
    dec   rcx
    call  l11
    pop   rsi
    pop   rdx
    mov   cl, 2
    ;compare user names
    ;then domain names
l5: lodsb
```

```
    scasb
    jne   l6
    test  al, al
    jne   l5
    mov   esi, ebx
    mov   edi, edx
    loop  l5
l6: pop   rdi
    pop   rsi
    pop   rbp
    pop   rbx
    jne   l8
    test  esi, esi
    je    l7
    mov   rsi, offset l17
    cmp   cl, [rsi]
    adc   [rsi], ecx
l7: mov   esi, [rbx+8] ;th32ProcessID
l8: push  rbx
    pop   rdx
    mov   ecx, ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    dec   b [offset l17+1]
    jne   l1
    jmp   being_debugged
l9: ...
l10:cdq
    xor   ecx, ecx
    mov   ch, 4 ;PROCESS_QUERY_INFORMATION
    enter 20h, 0
    call  OpenProcess
    leave
    xchg  ecx, eax
    jrcxz l14
l11:push  rax
    mov   r8d, esp
    push  8 ;TOKEN_QUERY
    pop   rdx
    call  OpenProcessToken
    pop   rbx
    xor   ebp, ebp
l12:mov   edx, ebp
```

```
    xor   ecx, ecx ;GMEM_FIXED
    enter 20h, 0
    call  GlobalAlloc
    leave
    push  rbp
    pop   r9
    push  rax
    pop   r8
    push  rax ;simulate enter
    mov   ebp, esp
    push  rbp
    sub   esp, 20h
    push  1 ;TokenUser
    pop   rdx
    mov   ecx, ebx
    xchg  esi, eax
    call  GetTokenInformation
    leave
    xchg  ecx, eax
    jrcxz l12
    xor   ebp, ebp
l13:mov   ebx, ebp
    mov   edx, ebp
    xor   ecx, ecx ;GMEM_FIXED
    enter 20h, 0
    call  GlobalAlloc
    xchg  ebx, eax
    xchg  edx, eax
    xor   ecx, ecx ;GMEM_FIXED
    call  GlobalAlloc
    leave
    xchg  edi, eax
    push  rbp
    mov   ecx, esp
    push  rbp
    mov   r9d, esp
    push  rax
    push  rsp
    push  rcx
    push  rbx
    sub   esp, 20h
    push  rdi
    pop   r8
    mov   edx, [rsi]
```

```
    xor   ecx, ecx
    call  LookupAccountSidA
    add   esp, 40h
    pop   rcx
    pop   rbp
    xchg  ecx, eax
    jrcxz l13
l14:ret
    ;trailing zero is converted to space
l15:db    "explorer.exe "
l16:dd    130h ;sizeof(PROCESSENTRY32)
    db    12ch dup (?)
l17:db    0ffh, 1, ?, ?
```

Note that this code is for demonstration purposes only. It demonstrates a number of poor programming practices, it leaks both handles and memory, and it is not intended to be used in any kind of product.

A variation of this technique looks for the names of particular tools that would suggest that a debugger or similar might be present. In this case, the presence of the tools in the sessions of other logged-on users might be acceptable. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push  0
    push  2 ;TH32CS_SNAPPROCESS
    call  CreateToolhelp32Snapshot
    mov   ebx, offset l6
    xchg  ebp, eax
l1: push  ebx
    push  ebp
    call  Process32First
l2: lea   ecx, [ebx+24h] ;szExeFile
    mov   esi, ecx
l3: lodsb
    cmp   al, "\"
    cmove ecx, esi
    or    b [esi-1], " "
    test  al, al
    jne   l3
```

```
    sub   esi, ecx
    xchg  ecx, esi
    mov   edi, offset l5
l4: push  ecx
    push  esi
    repe  cmpsb
    pop   esi
    pop   ecx
    je    being_debugged
    push  ecx
    or    ecx, -1
    repne scasb
    pop   ecx
    cmp   [edi], al
    jne   l4
    push  ebx
    push  ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    ...
l5: <array of ASCII strings
     space then zero to end each one
     zero to end the list
    >
l6: dd    128h ;sizeof(PROCESSENTRY32)
    db    124h dup (?)
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   edx, edx
    push  2 ;TH32CS_SNAPPROCESS
    pop   rcx
    call  CreateToolhelp32Snapshot
    mov   rbx, offset l6
    xchg  ebp, eax
l1: push  rbx
    pop   rdx
    mov   ecx, ebp
    call  Process32First
l2: lea   ecx, [rbx+2ch] ;szExeFile
    mov   esi, ecx
l3: lodsb
```

```
    cmp   al, "\"
    cmove ecx, esi
    or    b [rsi-1], " "
    test  al, al
    jne   l3
    sub   esi, ecx
    xchg  ecx, esi
    mov   rdi, offset l5
l4: push  rcx
    push  rsi
    repe  cmpsb
    pop   rsi
    pop   rcx
    je    being_debugged
    push  rcx
    or    ecx, -1
    repne scasb
    pop   rcx
    cmp   [rdi], al
    jne   l4
    push  rbx
    pop   rdx
    mov   ecx, ebp
    call  Process32Next
    test  eax, eax
    jne   l2
    ...
l5: <array of ASCII strings
     space then zero to end each one
     zero to end the list
    >
l6: dd    130h ;sizeof(PROCESSENTRY32)
    db    12ch dup (?)
```

Note that there would be a problem to detect a process whose name really has a space in it, since no attempt is made to distinguish between the real process name and a process whose name is the substring that ends exactly where the space appears. However, again, such a situation might be acceptable.

iv.DbgBreakPoint

The ntdll DbgBreakPoint() function is called when a debugger attaches to an already-running process. The function allows the debugger to gain control because an exception is raised that the debugger can intercept. However, this requires that the function remains intact. If the function is altered, it can be used to reveal the presence of the debugger, or to disallow the attaching. This attaching can be prevented by simply erasing the breakpoint. The patch can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push offset l1
    call GetModuleHandleA
    push offset l2
    push eax
    call GetProcAddress
    push eax
    push esp
    push 40h ;PAGE_EXECUTE_READWRITE
    push 1
    push eax
    xchg ebx, eax
    call VirtualProtect
    mov  b [ebx], 0c3h
    ...
l1: db   "ntdll", 0
l2: db   "DbgBreakPoint", 0
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov   rcx, offset l1
    call  GetModuleHandleA
    mov   rdx, offset l2
    xchg  rcx, rax
    call  GetProcAddress
    push  rax
    pop   rbx
    enter 20h, 0
    push  rbp
```

```
    pop   r9
    push  40h ;PAGE_EXECUTE_READWRITE
    pop   r8
    xor   edx, edx
    inc   edx
    xchg  rcx, rax
    call  VirtualProtect
    mov   b [rbx], 0c3h
    ...
l1: db   "ntdll", 0
l2: db   "DbgBreakPoint", 0
```

v.DbgPrint

Normally, the ntdll DbgPrint() function raises an exception, but a registered Structured Exception Handler will not see it. The reason is that *Windows* registers its own Structured Exception Handler internally. This handler will consume the exception if a debugger does not do so. "Normally" in this case refers to the fact that the exception can be suppressed on *Windows 2000* and later.

On *Windows NT*, there is a problem if the ntdll _vsnprintf() function is hooked and also calls the ntdll DbgPrint() function. The result in that case would be a recursive call until a stack overflow occurred. This bug was fixed in *Windows 2000*, by adding a "busy" flag at offset 0xf74 in the Thread Environment Block. The function will exit immediately if the flag is set. However, the bug was reintroduced in the ntdll DbgPrintReturnControlC() function that was added in *Windows 2000*, and it is also present in *Windows XP*. The bug was fixed in *Windows Vista*, by adding a "busy" flag in bit 1 at offset 0xfca in the Thread Environment Block (but also removing the busy flag at offset 0xf74 in the Thread Environment Block).

Windows XP introduced the ntdll DbgPrintEx() function and ntdll vDbgPrintEx() functions, which extended the behaviour even further. Both functions accept the severity level as a parameter. If the value is not 0xffffffff, then the ntdll NtQueryDebugFilterState() function is called with the component ID and the severity level. If the corresponding entry is zero in the kernel debugger table, then the function returns immediately.

There is a more serious bug in *Windows XP*, which is that the ntdll vDbgPrintExWithPrefix() function does not check the length of the prefix while copying it to a stack buffer. If the string is long enough, then the return address (and optionally the Structured Exception Handler record) can be replaced, potentially resulting in the execution of arbitrary code. However, the major mitigating factor here is that this is an internal function, which should not be called directly. The public functions which call the ntdll vDbgPrintExWithPrefix() function all use an empty prefix. Prior to *Windows XP*, a fixed-length copy was performed. Since this was unnecessarily large in most cases, and had the potential to cause crashes because of out-of-bounds reads, it was replaced in *Windows XP* with a string copy. However, the length of the string was not verified prior to performing the copy, leading to the buffer overflow. This kind of bug is known to exist in at least one other DLL in *Windows XP*.

Windows Vista changed the behaviour yet again. If a debugger is present and the severity level is 0x65, then the ntdll NtQueryDebugFilterState() function is not called. If the severity level is not 0xffffffff, and a debugger is not present or the severity level is not 0x65, then the ntdll NtQueryDebugFilterState() function will be called with the component ID and severity level. As before, if the corresponding entry is zero in the kernel debugger table, then the function returns immediately. The busy flag is checked only at this point, and the function returns immediately if the flag is set.

If a user-mode debugger is present (which is determined by reading the BeingDebugged flag in the

Process Environment Block), or a kernel-mode debugger is not present in *Windows Vista* and later (which is determined by reading the KdDebuggerEnabled member of the KUSER_SHARED_DATA structure, at offset 0x7ffe02d4 for 2Gb user-space configurations), then *Windows* will raise the DBG_PRINTEXCEPTION_C (0x40010006) exception. If no user-mode debugger is present, and if a kernel-mode debugger is present in *Windows Vista* and later, then *Windows* will execute an interrupt 0x2d (which does not result in an exception) and then return.

In *Windows XP* and later, if an exception occurs, any registered Vectored Exception Handler will run before the Structured Exception Handler that *Windows* registers. This might be considered a bug in *Windows*. If nothing else, it is a curious oversight, but ultimately, all of these details just mean that the presence of a debugger cannot be inferred if the exception is not delivered to the Vectored Exception Handler.

vi.DbgSetDebugFilterState

The ntdll DbgSetDebugFilterState() function (or the ntdll NtSetDebugFilterState() function, both introduced in *Windows XP*) *cannot* be used to detect the presence of a debugger, despite what its name suggests, because it simply sets a flag in a table that would be checked by a kernel-mode debugger, if it were present. While it is true that the function call will succeed in the presence of some debuggers, this is due to a side-effect of the debugger's behaviour (specifically, acquiring the debug privilege), and not due to the debugger itself (this should be obvious since the function call does not succeed when used with certain debuggers). All it reveals is that the user account for the process is a member of the administrators group and it has the debug privilege. The reason is that the success or failure of the function call is limited only by the process privilege level. If the user account of the process is a member of the administrators group and has the debug privilege, then the function call will succeed; if not, then not. It is not sufficient for a standard user to acquire the debug privilege, nor can an administrator call the function successfully without it. The filter call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
push  1
push  0
push  0
call  NtSetDebugFilterState
xchg  ecx, eax
jecxz admin_with_debug_priv
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
push  1
pop   r8
xor   edx, edx
xor   ecx, ecx
call  NtSetDebugFilterState
xchg  ecx, eax
jecxz admin_with_debug_priv
```

vii.IsDebuggerPresent

The kernel32 IsDebuggerPresent() function was introduced in *Windows 95*. It returns a non-zero value if a debugger is present. The check can be made using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit *Windows* environment:

```
call IsDebuggerPresent
test al, al
jne  being_debugged
```

Internally, the function simply returns the value of the BeingDebugged flag. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
mov eax, fs:[30h] ;Process Environment Block
cmp b [eax+2], 0 ;check BeingDebugged
jne being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
push 60h
pop  rsi
gs:lodsq ;Process Environment Block
cmp b [rax+2], 0 ;check BeingDebugged
jne being_debugged
```

or using this 32-bit code to examine the 64-bit *Windows* environment:

```
mov eax, fs:[30h] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
cmp b [eax+1002h], 0 ;check BeingDebugged
jne being_debugged
```

To defeat these methods requires only setting the BeingDebugged flag to zero.

viii.NtQueryInformationProcess

a.ProcessDebugPort

The ntdll NtQueryInformationProcess() function accepts a parameter which is the class of information to query. Most of the classes are not documented. However, one of the documented classes is the ProcessDebugPort (7). It is possible to query for the existence (not the value) of the port. The return value is 0xffffffff if the process is being debugged. Internally, the function queries for the non-zero state of the DebugPort field in the EPROCESS structure. Internally, this is how the kernel32 CheckRemoteDebuggerPresent() function works. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 7 ;ProcessDebugPort
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
inc  eax
je   being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
xor   ebp, ebp
enter 20h, 0
push  8 ;ProcessInformationLength
pop   r9
push  rbp
pop   r8
push  7 ;ProcessDebugPort
pop   rdx
or    rcx, -1 ;GetCurrentProcess()
call  NtQueryInformationProcess
leave
test  ebp, ebp
jne  being_debugged
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

b. ProcessDebugObjectHandle

Windows XP introduced a "debug object". When a debugging session begins, a debug object is created, and a handle is associated with it. It is possible to query for the value of this handle, using the undocumented ProcessDebugObjectHandle (0x1e) class. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 1eh ;ProcessDebugObjectHandle
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
jne  being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
xor   ebp, ebp
enter 20h, 0
push  8 ;ProcessInformationLength
pop   r9
push  rbp
pop   r8
push  1eh ;ProcessDebugObjectHandle
pop   rdx
or    rcx, -1 ;GetCurrentProcess()
call  NtQueryInformationProcess
leave
test  ebp, ebp
jne   being_debugged
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger. Note that querying the debug object handle on a 64-bit system, in the presence of a debugger, can increase the handle's reference count, thus preventing the debuggee from terminating.

c. ProcessDebugFlags

The undocumented ProcessDebugFlags (0x1f) class returns the inverse value of the NoDebugInherit bit in the EPROCESS structure. That is, the return value is zero if a debugger is present. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 1fh ;ProcessDebugFlags
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
je   being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
xor   ebp, ebp
enter 20h, 0
push  4 ;ProcessInformationLength
pop   r9
push  rbp
pop   r8
push  1fh ;ProcessDebugFlags
pop   rdx
or    rcx, -1 ;GetCurrentProcess()
call  NtQueryInformationProcess
leave
test  ebp, ebp
je   being_debugged
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

ix.OutputDebugString

The kernel32 OutputDebugString() function can demonstrate different behaviour, depending on the version of *Windows*, and whether or not a debugger is present. The most obvious difference in behaviour that the kernel32 GetLastError() function will return zero if a debugger is present, and non-zero if a debugger is not present. However, this applies only to *Windows NT/2000/XP*. On *Windows Vista* and later, the error code is unchanged in all cases. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  ebp, ebp
    push ebp
    push esp
    call OutputDebugStringA
    cmp  fs:[ebp+34h], ebp ;LastErrorValue
    je   being_debugged
```

or this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   ebp, ebp
    enter 20h, 0
    mov   ecx, ebp
    call  OutputDebugStringA
    cmp   gs:[rbp+68h], ebp ;LastErrorValue
    je    being_debugged
```

The reason why it worked was that *Windows* attempted to open a mapping to an object called "DBWIN_BUFFER". When it failed, the error code is set. Following that was a call to the ntdll DbgPrint() function. As noted above, if a debugger is present, the exception might be consumed by the debugger, resulting in the error code being cleared. If no debugger is present, then the exception would be consumed by *Windows*, and the error code would remain. However, in *Windows Vista* and later, the error code is restored to the value that it had prior to the kernel32 OutputDebugString() function being called. It is not cleared explicitly, resulting in this detection technique becoming completely unreliable.

The function is perhaps most well-known because of a bug in *OllyDbg* v1.10 that results from its use. *OllyDbg* passes user-defined data directly to the msvcrt _vsprintf() function. Those data can contain string-formatting tokens. A specific token in a specific position will cause the function to attempt to access memory using one of the passed parameters. A number of variations of the attack exist, all of which are essentially randomly chosen token combinations that happen to work. However, all that is required is three tokens. The first two tokens are entirely arbitrary. The third token must be a "%s". This is because the _vsprintf() function calls the __vprinter() function, and passes a zero as the fourth parameter. The fourth parameter is accessed by the third token, if the "%s" is used there. The result is a null-pointer access, and a crash. The bug cannot be exploited to execute arbitrary code.

x.RtlQueryProcessHeapInformation

The ntdll RtlQueryProcessHeapInformation() function can be used to read the heap flags from the process memory of the current process. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is (or might be) within the 3.10-3.50 range:

```
    push 0
    push 0
    call RtlCreateQueryDebugBuffer
    push eax
    xchg ebx, eax
    call RtlQueryProcessHeapInformation
    mov  eax, [ebx+38h] ;HeapInformation
    mov  eax, [eax+8] ;Flags
    ;neither CREATE_ALIGN_16
```

```
;nor HEAP_SKIP_VALIDATION_CHECKS
and  eax, 0effeffffh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp  eax, 40000062h
je   being_debugged
```

or this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is 3.51 or greater:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
call  RtlQueryProcessHeapInformation
mov   eax, [ebx+38h] ;HeapInformation
mov   eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
and   al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp   eax, 62000040h
je    being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
xor  edx, edx
xor  ecx, ecx
call RtlCreateQueryDebugBuffer
mov  ebx, eax
xchg ecx, eax
call RtlQueryProcessHeapInformation
mov  eax, [rbx+70h] ;HeapInformation
;Flags
;GROWABLE
```

```
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp  d [rax+10h], 40000062h
je   being_debugged
```

xi.RtlQueryProcessDebugInformation

The ntdll RtlQueryProcessDebugInformation() function can be used to read certain fields from the process memory of the requested process, including the heap flags. The function does this for the heap flags by calling the ntdll RtlQueryProcessHeapInformation() function internally. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is (or might be) within the 3.10-3.50 range:

```
xor  ebx, ebx
push ebx
push ebx
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
push 14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
push d fs:[eax+20h] ;UniqueProcess
call RtlQueryProcessDebugInformation
mov  eax, [ebx+38h] ;HeapInformation
mov  eax, [eax+8] ;Flags
;neither CREATE_ALIGN_16
;nor HEAP_SKIP_VALIDATION_CHECKS
and  eax, 0effeffffh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp  eax, 40000062h
je   being_debugged
```

or this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*, if the subsystem version is 3.51 or greater:

```
xor  ebx, ebx
push ebx
push ebx
call  RtlCreateQueryDebugBuffer
push  eax
push  14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
xchg  ebx, eax
push  d fs:[eax+20h] ;UniqueProcess
call  RtlQueryProcessDebugInformation
mov   eax, [ebx+38h] ;HeapInformation
mov   eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
and   al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
;reversed by bswap
cmp   eax, 62000040h
je    being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
xor  edx, edx
xor  ecx, ecx
call RtlCreateQueryDebugBuffer
push rax
pop  r8
push 14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
pop  rdx
mov  ecx, gs:[rdx+2ch] ;UniqueProcess
xchg ebx, eax
call RtlQueryProcessDebugInformation
mov  eax, [rbx+70h] ;HeapInformation
;Flags
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
```

```
    cmp  d [rax+10h], 40000062h
    je   being_debugged
```

xii. SwitchToThread

The kernel32 SwitchToThread() function (or the ntdll NtYieldExecution() function) allows the current thread to offer to give up the rest of its time slice, and allow the next scheduled thread to execute. If no threads are scheduled to execute (or when the system is busy in particular ways and will not allow a switch to occur), then the ntdll NtYieldExecution() function returns the STATUS_NO_YIELD_PERFORMED (0x40000024) status, which causes the kernel32 SwitchToThread() function to return a zero. When an application is being debugged, the act of single-stepping through the code causes debug events and often results in no yield being allowed. However, this is a hopelessly unreliable method for detecting a debugger because it will also detect the presence of a thread that is running with high priority. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push  20h
    pop   ebp
l1: push  0fh
    call  Sleep
    call  SwitchToThread
    cmp   al, 1
    adc   ebx, ebx
    dec   ebp
    jne   l1
    inc   ebx ;detect 32 non-yields
    je    being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    push  20h
    pop   rbp
```

```
l1: push  0fh
    pop   rcx
    call  Sleep
    call  SwitchToThread
    cmp   al, 1
    adc   ebx, ebx
    dec   ebp
    jne   l1
    inc   ebx ;detect 32 non-yields
    je    being_debugged
```

xiii.Toolhelp32ReadProcessMemory

The kernel32 Toolhelp32ReadProcessMemory() function (introduced in *Windows 2000*) allows one process to open and read the memory of another process. It combines the kernel32 OpenProcess() function with the kernel32 ReadProcessMemory() function (or the kernel32 CloseHandle() function). Note that the function is currently documented incorrectly regarding how to copy data from the current process. The function is documented as accepting a zero for the process ID[3] to read the memory of the current process but this is incorrect. The process ID must always be valid. To read from the current process, the process ID must be the value that is returned by the kernel32 GetCurrentProcessId() function. This function can be used as another way to detect a step-over condition, by checking for a breakpoint after the function call. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
mov  eax, esp
xor  ebx, ebx
push ebx
inc  ebx
push ebx
push eax
```

[3] http://msdn.microsoft.com/en-us/library/ms686826(VS.85).aspx

```
    push offset l1
    push d fs:[ebx+1fh] ;UniqueProcess
    call Toolhelp32ReadProcessMemory
l1: pop  eax
    cmp  al, 0cch
    je   being_debugged
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    xor   ebp, ebp
    enter 20h, 0
    push  1
    pop   r9
    push  rbp
    pop   r8
    mov   rdx, offset l1
    mov   ecx, gs:[rbp+40h] ;UniqueProcess
    call  Toolhelp32ReadProcessMemory
l1: leave
    cmp   bpl, 0cch
    je    being_debugged
```

xiv.UnhandledExceptionFilter

When an exception occurs, and no registered Exception Handlers exist (neither Structured nor Vectored), or if none of the registered handlers handles the exception, then the kernel32 UnhandledExceptionFilter() function will be called as a last resort. If no debugger is present (which is determined by calling the ntdll NtQueryInformationProcess() function with the ProcessDebugPort class), then the handler will be called that was registered by the kernel32 SetUnhandledExceptionFilter() function. If a debugger is present, then that call will not be reached. Instead, the exception will be passed to the debugger. The function determines the presence of a debugger by calling the ntdll NtQueryInformationProcess function with the ProcessDebugPort class. The missing exception can be used to infer the presence of the debugger. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push offset l1
    call SetUnhandledExceptionFilter
    ;force an exception to occur
    int  3
    jmp  being_debugged
l1: ;execution resumes here if exception occurs
    ...
```

or using this 64-bit code on the 64-bit versions of *Windows* (but the technique does not work in the same way for 64-bit processes):

```
    mov  rcx, offset l1
    call SetUnhandledExceptionFilter
    ;force an exception to occur
    int  3
    jmp  being_debugged
l1: ;execution resumes here if exception occurs
    ...
```

xv.VirtualProtect

The kernel32 VirtualProtect() function (or the kernel32 VirtualProtectEx() function, or then ntdll NtProtectVirtualMemory() function) can be used to allocate "guard" pages. Guard pages are pages that trigger an exception the first time that they are accessed. They are commonly placed at the bottom of a stack, to intercept a potential problem before it becomes unrecoverable. Guard pages can also be used to detect a debugger. The two preliminary steps are to register an exception handler, and to allocate the guard page. The order of these steps is not important. Typically, the page is allocated initially as writable and executable, to allow some content to be placed in it, though this is entirely optional. After filling the page, the page protections are altered to convert the page to a guard page. The next step is to attempt to execute something from the guard page. This should result in an EXCEPTION_GUARD_PAGE (0x80000001) exception being received by the exception handler. However, if a debugger is present, then the debugger might intercept the exception and allow the execution to continue. This behaviour is known to occur in *OllyDbg*. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  ebx, ebx
    push 40h ;PAGE_EXECUTE_READWRITE
    push 1000h ;MEM_COMMIT
    push 1
    push ebx
    call VirtualAlloc
    mov  b [eax], 0c3h
    push eax
    push esp
    push 140h ;PAGE_EXECUTE_READWRITE+PAGE_GUARD
    push 1
    push eax
    xchg ebp, eax
    call VirtualProtect
    push offset l1
    push d fs:[ebx]
    mov  fs:[ebx], esp
    push offset being_debugged
    ;execution resumes at being_debugged
    ;if ret instruction is executed
    jmp  ebp
l1: ;execution resumes here if exception occurs
    ...
```

or using this 64-bit code on the 64-bit versions of *Windows* (though this would apply to debuggers other than *OllyDbg* because *OllyDbg* does not run on the 64-bit versions of *Windows*):

```
    push  40h ;PAGE_EXECUTE_READWRITE
    pop   r9
    mov   r8d, 1000h ;MEM_COMMIT
    push  1
    pop   rdx
    xor   ecx, ecx
```

```
     call  VirtualAlloc
     mov   b [rax], 0c3h
     push  rax
     pop   rbx
     enter 20h, 0
     push  rbp
     pop   r9
     ;PAGE_EXECUTE_READWRITE+PAGE_GUARD
     mov   r8d, 140h
     push  1
     pop   rdx
     xchg  rcx, rax
     call  VirtualProtect
     mov   rdx, offset l1
     xchg  ecx, eax
     call  AddVectoredExceptionHandler
     push  offset being_debugged
     ;execution resumes at being_debugged
     ;if ret instruction is executed
     jmp   rbx
l1:  ;execution resumes here if exception occurs
     ...
```

E.System-level

    FindWindow
    NtQueryObject
    NtQuerySystemInformation

i.FindWindow

The user32 FindWindow() function can be used to search for windows by name or class. This is an easy way to detect the presence of a debugger, if the debugger has a graphical user interface. For example, *OllyDbg* can be found by passing "OLLYDBG" as the class name to find. *WinDbg* can be found by passing "WinDbgFrameClass" as the class name to find. The presence of these tools can be checked using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    mov   edi, offset l2
l1: push  0
    push  edi
    call  FindWindowA
    test  eax, eax
    jne   being_debugged
    or    ecx, -1
    repne scasb
    cmp   [edi], al
    jne   l1
    ...
l2: <array of ASCIIZ strings, zero to end>
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  rdi, offset l2
l1: xor  edx, edx
    push rdi
    pop  rcx
    call FindWindowA
    test eax, eax
    jne  being_debugged
    or    ecx, -1
    repne scasb
    cmp   [rdi], al
    jne   l1
    ...
l2: <array of ASCIIZ strings, zero to end>
```

A typical list includes the following names:

```
    db   "OLLYDBG", 0
    db   "WinDbgFrameClass", 0 ;WinDbg
    db   "ID", 0 ;Immunity Debugger
    db   "Zeta Debugger", 0
    db   "Rock Debugger", 0
    db   "ObsidianGUI", 0
    db   0
```

ii.NtQueryObject

Windows XP introduced a "debug object". When a debugging session begins, a debug object is created, and a handle is associated with it. Using the ntdll NtQueryObject() function, it is possible to query for the list of existing objects, and check the number of handles associated with any debug object that exists (the function can be called on any *Windows NT*-based platform, but only *Windows XP* and later will have a debug object in the list). The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor   ebx, ebx
    xor   ebp, ebp
    jmp   l2
l1: push  8000h ;MEM_RELEASE
    push  ebp
    push  esi
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   ebx, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    push  eax
    push  ebx
    push  ebp
    call  VirtualAlloc
    ;function does not return required length
    ;for this class in Windows Vista and later
    push  ebp
    ;must calculate by brute-force
    push  ebx
    push  eax
    push  3 ;ObjectAllTypesInformation
    push  ebp
    xchg  esi, eax
    call  NtQueryObject
    ;presumably STATUS_INFO_LENGTH_MISMATCH
    test  eax, eax
    jl    l1
```

```
    lodsd ;handle count
    xchg  ecx, eax
l3: lodsd ;string lengths
    movzx edx, ax ;length
    lodsd ;pointer to TypeName
    xchg  esi, eax
    ;sizeof(L"DebugObject")
    ;avoids superstrings
    ;like "DebugObjective"
    cmp   edx, 16h
    jne   l4
    xchg  ecx, edx
    mov   edi, offset l5
    repe  cmpsb
    xchg  ecx, edx
    jne   l4
    ;checking TotalNumberOfHandles
    ;works only on Windows XP
    ;cmp   [eax], edx ;TotalNumberOfHandles
    ;check TotalNumberOfObjects instead
    cmp   [eax+4], edx ;TotalNumberOfObjects
    jne   being_debugged
l4: lea   esi, [esi+edx+4] ;skip null and align
    and   esi, -4 ;round down to dword
    loop  l3
    ...
l5: dw    "D","e","b","u","g"
    dw    "O","b","j","e","c","t"
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   ebx, ebx
    xor   ebp, ebp
    jmp   l2
l1: mov   r8d, 8000h ;MEM_RELEASE
    xor   edx, edx
    mov   ecx, esi
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   ebx, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    pop   r9
```

```
    push  rax
    pop   r8
    mov   edx, ebx
    xor   ecx, ecx
    call  VirtualAlloc
    ;function does not return required length
    ;for this class in Windows Vista and later
    enter 20h, 0
    ;must calculate by brute-force
    push  rbx
    pop   r9
    push  rax
    pop   r8
    push  3 ;ObjectAllTypesInformation
    pop   rdx
    xor   ecx, ecx
    xchg  esi, eax
    call  NtQueryObject
    leave
    ;presumably STATUS_INFO_LENGTH_MISMATCH
    test  eax, eax
    jl    l1
    lodsq ;handle count
    xchg  ecx, eax
l3: lodsq ;string lengths
    movzx edx, ax ;length
    lodsq ;pointer to TypeName
    xchg  esi, eax
    ;sizeof(L"DebugObject")
    ;avoids superstrings
    ;like "DebugObjective"
    cmp   edx, 16h
    jne   l4
    xchg  ecx, edx
    mov   rdi, offset l5
    repe  cmpsb
    xchg  ecx, edx
    jne   l4
    ;checking TotalNumberOfHandles
    ;works only on Windows XP
    ;cmp   [rax], edx ;TotalNumberOfHandles
    ;check TotalNumberOfObjects instead
    cmp   [rax+4], edx ;TotalNumberOfObjects
    jne   being_debugged
```

```
l4: lea   esi, [rsi+rdx+8] ;skip null and align
    and   esi, -8 ;round down to dword
    loop  l3
    ...
l5: dw    "D","e","b","u","g"
    dw    "O","b","j","e","c","t"
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

iii.NtQuerySystemInformation

a.SystemKernelDebuggerInformation

The ntdll NtQuerySystemInformation() function accepts a parameter which is the class of information to query. Most of the classes are not documented. This includes the SystemKernelDebuggerInformation (0x23) class, which has existed since *Windows NT*. The SystemKernelDebuggerInformation class returns the value of two flags: KdDebuggerEnabled in al, and KdDebuggerNotPresent in ah. Thus, the return value in ah is zero if a debugger *is* present. The call can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
push eax
mov  eax, esp
push 0
push 2 ;SystemInformationLength
push eax
push 23h ;SystemKernelDebuggerInformation
call NtQuerySystemInformation
pop  eax
test ah, ah
je   being_debugged
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
push rax
mov  edx, esp
xor  r9, r9
push 2 ;SystemInformationLength
pop  r8
push 23h ;SystemKernelDebuggerInformation
pop  rcx
call NtQuerySystemInformation
pop  rax
test ah, ah
je   being_debugged
```

Since this information comes from the kernel, there is no easy way to prevent this call from revealing the presence of the debugger. The function writes only two bytes, regardless of the input size. This small size is unusual, and can reveal the presence of some hiding tools, because such tools typically write four bytes to the destination.

The function call can be further obfuscated by simply retrieving the value directly from the KdDebuggerEnabled field in the KUSER_SHARED_DATA structure, at offset 0x7ffe02d4 for 2Gb user-space configurations. This value is available on all 32-bit and 64-bit versions of *Windows*. Interestingly, the value that is returned by the function call comes from a separate location, so any tool that wants to hide the debugger would need to patch the value in both locations.

b.SystemProcessInformation

The process ID of both Explorer.exe and the parent process of the current process, and the name of that parent process, can be obtained by the ntdll NtQuerySystemInformation() function with the SystemProcessInformation (5) class. A single call to the function returns the entire list of running processes, which must then be parsed manually. This is the function that the kernel32 CreateToolhelp32Snapshot() function calls internally. As with the kernel32 CreateToolhelp32Snapshot() function algorithm, the user-name and domain name should be checked to avoid false matching and to potentially reduce the number of passes of the routine. The call can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor   ebp, ebp
    xor   esi, esi
    xor   edi, edi
    jmp   l2
l1: push  8000h ;MEM_RELEASE
    push  ebp
    push  ebx
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   esi, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    push  eax
    push  esi
    push  ebp
    call  VirtualAlloc
    ;function does not return
    ;required length for this class
    push  ebp
    ;must calculate by brute-force
    push  esi
    push  eax
    push  5 ;SystemProcessInformation
    xchg  ebx, eax
    call  NtQuerySystemInformation
    ;presumably STATUS_INFO_LENGTH_MISMATCH
    test  eax, eax
    jl    l1
    push  ebx
    push  ebx
l3: push  ebx
    mov   eax, fs:[20h] ;UniqueProcess
    cmp   [ebx+44h], eax ;UniqueProcessId
    ;InheritedFromUniqueProcessId
    cmove edi, [ebx+48h]
```

```
    test  edi, edi
    je    l4
    cmp   ebp, edi
    je    l11
l4: mov   ecx, [ebx+3ch] ;ImageName
    jecxz l9
    xor   eax, eax
    mov   esi, ecx
l5: lodsw
    cmp   eax, "\"
    cmove ecx, esi
    push  ecx
    push  eax
    call  CharLowerW
    mov   [esi-2], ax
    pop   ecx
    test  eax, eax
    jne   l5
    sub   esi, ecx
    xchg  ecx, esi
    push  edi
    mov   edi, offset l17
    repe  cmpsb
    pop   edi
    jne   l9
    mov   eax, [ebx+44h] ;UniqueProcessId
    push  ebx
    push  ebp
    push  edi
    call  l12
    dec   ecx ;invert Z flag
    jne   l7
    push  ebx
    push  edi
    dec   ecx
    call  l13
    pop   esi
    pop   edx
    mov   cl, 2
    ;compare user names
    ;then domain names
l6: lodsb
    scasb
    jne   l7
```

```
    test  al, al
    jne   l6
    mov   esi, ebx
    mov   edi, edx
    loop  l6
l7: pop   edi
    pop   ebp
    pop   ebx
    jne   l9
    test  ebp, ebp
    je    l8
    mov   esi, offset l18
    cmp   cl, [esi]
    adc   [esi], ecx
l8: mov   ebp, [ebx+44h] ;UniqueProcessId
l9: pop   ebx
    mov   ecx, [ebx] ;NextEntryOffset
    add   ebx, ecx
    inc   ecx
    loop  l10
    pop   ebx
    dec   b [offset l18+1]
l10:jne   l3
    ;and possibly one pointer left on stack
    ;add esp, -b [offset l18]*4
    jmp   being_debugged
    ;and at least one pointer left on stack
    ;add esp, (b [offset l18+1]-b [offset l18]+1)*4
l11:...
l12:push  eax
    push  0
    push  400h ;PROCESS_QUERY_INFORMATION
    call  OpenProcess
    xchg  ecx, eax
    jecxz l16
l13:push  eax
    push  esp
    push  8 ;TOKEN_QUERY
    push  ecx
    call  OpenProcessToken
    pop   ebx
    xor   ebp, ebp
l14:push  ebp
    push  0 ;GMEM_FIXED
```

```
    call  GlobalAlloc
    push  eax
    push  esp
    push  ebp
    push  eax
    push  1 ;TokenUser
    push  ebx
    xchg  esi, eax
    call  GetTokenInformation
    pop   ebp
    xchg  ecx, eax
    jecxz l14
    xor   ebp, ebp
l15:push  ebp
    push  0 ;GMEM_FIXED
    call  GlobalAlloc
    xchg  ebx, eax
    push  ebp
    push  0 ;GMEM_FIXED
    call  GlobalAlloc
    xchg  edi, eax
    push  eax
    mov   eax, esp
    push  ebp
    mov   ecx, esp
    push  ebp
    mov   edx, esp
    push  eax
    push  ecx
    push  ebx
    push  edx
    push  edi
    push  d [esi]
    push  0
    call  LookupAccountSidA
    pop   ebp
    pop   ecx
    xchg  ebp, ecx
    pop   edx
    xchg  ecx, eax
    jecxz l15
l16:ret
l17:dw    "e", "x", "p", "l", "o", "r", "e", "r"
    dw    ".", "e", "x", "e", 0
```

```
l18:db    0ffh, 1, ?, ?
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor   ebp, ebp
    xor   esi, esi
    xor   edi, edi
    jmp   l2
l1: mov   r8d, 8000h ;MEM_RELEASE
    xor   edx, edx
    mov   ecx, ebx
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   esi, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    pop   r9
    push  rax
    pop   r8
    mov   edx, esi
    xor   ecx, ecx
    call  VirtualAlloc
    ;function does not return
    ;required length for this class
    xor   r9d, r9d
    ;must calculate by brute-force
    push  rsi
    pop   r8
    mov   edx, eax
    push  5 ;SystemProcessInformation
    pop   rcx
    xchg  ebx, eax
    call  NtQuerySystemInformation
    ;presumably STATUS_INFO_LENGTH_MISMATCH
    test  eax, eax
    jl    l1
    push  rbx
    push  rbx
l3: push  rbx
    call  GetCurrentProcessId
    cmp   [rbx+50h], eax ;UniqueProcessId
    ;InheritedFromUniqueProcessId
    cmove edi, [rbx+58h]
```

```
    test  edi, edi
    je    l4
    cmp   ebp, edi
    je    l11
l4: mov   ecx, [rbx+40h] ;ImageName
    jrcxz l9
    xor   eax, eax
    mov   esi, ecx
l5: lodsw
    cmp   eax, "\"
    cmove ecx, esi
    push  rcx
    xchg  ecx, eax
    enter 20h, 0
    call  CharLowerW
    leave
    mov   [rsi-2], ax
    pop   rcx
    test  eax, eax
    jne   l5
    sub   esi, ecx
    xchg  ecx, esi
    push  rdi
    mov   rdi, offset l17
    repe  cmpsb
    pop   rdi
    jne   l9
    mov   r8d, [rbx+50h] ;UniqueProcessId
    push  rbx
    push  rbp
    push  rdi
    call  l12
    dec   ecx ;invert Z flag
    jne   l7
    push  rbx
    push  rdi
    dec   rcx
    call  l13
    pop   rsi
    pop   rdx
    inc   ecx
    ;compare user names
    ;then domain names
l6: lodsb
```

```
    scasb
    jne   l7
    test  al, al
    jne   l6
    mov   esi, ebx
    mov   edi, edx
    loop  l6
l7: pop   rdi
    pop   rbp
    pop   rbx
    jne   l9
    test  ebp, ebp
    je    l8
    mov   rsi, offset l18
    cmp   cl, [rsi]
    adc   [rsi], ecx
l8: mov   ebp, [rbx+50h] ;UniqueProcessId
l9: pop   rbx
    mov   ecx, [rbx] ;NextEntryOffset
    add   ebx, ecx
    inc   ecx
    loop  l10
    pop   rbx
    dec   byte [offset l18+1]
l10:jne   l3
    ;and possibly one pointer left on stack
    ;add esp, -b [offset l18]*4
    jmp   being_debugged
    ;and at least one pointer left on stack
    ;add esp, (b [offset l18+1]-b [offset l18]+1)*4
l11:...
l12:cdq
    xor   ecx, ecx
    mov   ch, 4 ;PROCESS_QUERY_INFORMATION
    enter 20h, 0
    call  OpenProcess
    leave
    xchg  ecx, eax
    jrcxz l16
l13:push  rax
    mov   r8d, esp
    push  8 ;TOKEN_QUERY
    pop   rdx
    call  OpenProcessToken
```

```
    pop   rbx
    xor   ebp, ebp
l14:mov   edx, ebp
    xor   ecx, ecx ;GMEM_FIXED
    enter 20h, 0
    call  GlobalAlloc
    leave
    push  rbp
    pop   r9
    push  rax
    pop   r8
    push  rax ;simulate enter
    mov   ebp, esp
    push  rbp
    sub   esp, 20h
    push  1 ;TokenUser
    pop   rdx
    mov   ecx, ebx
    xchg  esi, eax
    call  GetTokenInformation
    leave
    xchg  ecx, eax
    jrcxz l14
    xor   ebp, ebp
l15:mov   ebx, ebp
    mov   edx, ebp
    xor   ecx, ecx ;GMEM_FIXED
    enter 20h, 0
    call  GlobalAlloc
    xchg  ebx, eax
    xchg  edx, eax
    xor   ecx, ecx ;GMEM_FIXED
    call  GlobalAlloc
    leave
    xchg  edi, eax
    push  rbp
    mov   ecx, esp
    push  rbp
    mov   r9d, esp
    push  rax
    push  rsp
    push  rcx
    push  rbx
    sub   esp, 20h
```

```
    push  rdi
    pop   r8
    mov   edx, [rsi]
    xor   ecx, ecx
    call  LookupAccountSidA
    add   esp, 40h
    pop   rcx
    pop   rbp
    xchg  ecx, eax
    jrcxz l15
l16:ret
l17:dw    "e","x","p","l","o","r","e","r"
    dw    ".","e","x","e",0
l18:db    0ffh, 1, ?, ?
```

iv.Selectors

Selector values might appear to be stable, but they are actually volatile in certain circumstances, and also depending on the version of *Windows*. For example, a selector value can be set within a thread, but it might not hold that value for very long. Certain events might cause the selector value to be changed back to its default value. One such event is an exception. In the context of a debugger, the single-step exception is still an exception, which can cause some unexpected behaviour. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push fs
    pop  ds
l1: xchg [eax], cl
    xchg [eax], cl
```

There is no 64-bit code example because the DS selector is not supported in that environment.

On the 64-bit versions of *Windows*, single-stepping through this code will cause an access violation exception at l1 because the DS selector will be restored to its default value even before l1 is reached. On the 32-bit versions of *Windows*, the DS selector will not have its value restored, unless a non-debugging exception occurs. The version-specific difference in behaviours expands even further if the SS selector is used. On the 64-bit versions of *Windows*, the SS selector will be restored to its default value, as in the DS selector case. However, on the 32-bit versions of *Windows*, the SS selector value will not be restored, *even if an exception occurs*. Thus, if we change the code to look like this:

```
    xor  eax, eax
    push offset l2
    push d fs:[eax]
    mov  fs:[eax], esp
    push fs
    pop  ss
    xchg [eax], cl
    xchg [eax], cl
l1: int  3 ;force exception to occur
l2: ;looks like it would be reached
    ;if an exception occurs
    ...
```

then when the "int 3" instruction is reached at l1 and the breakpoint exception occurs, the exception handler at l2 is not called as expected. Instead, the process is simply terminated.

A variation of this technique detects the single-step event by simply checking if the assignment was successful. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push 3
    pop  gs
    mov  ax, gs
    cmp  al, 3
    jne  being_debugged
```

The FS and GS selectors are special cases. For certain values, they will be affected by the single-step event, even on the 32-bit versions of *Windows*. However, in the case of the FS selector (and, technically, the GS selector), it will be not restored to its default value on the 32-bit versions of *Windows*, if it was set to a value from zero to three. Instead, it will be set to zero (the GS selector is affected in the same way, but the default value for the GS selector is zero). On the 64-bit versions of *Windows*, it (they) will be restored to its (their) default value.

This code is also vulnerable to a race condition caused by a thread-switch event. When a thread-switch event occurs, it behaves like an exception, and will cause the selector values to be altered, which, in the case of the FS selector, means that it will be set to zero.

A variation of this technique solves that problem by waiting intentionally for a thread-switch event to occur. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push 3
    pop  gs
l1: mov  ax, gs
    cmp  al, 3
    je   l1
```

However, this code is vulnerable to the problem that it was trying to detect in the first place, because it does not check if the original assignment was successful. Of course, the two code snippets can be combined to produce the desired effect, by waiting until the thread-switch event occurs, and then performing the assignment within the window of time that should exist until the next one occurs. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push 3
    pop  gs
l1: mov  ax, gs
    cmp  al, 3
    je   l1
    push 3
    pop  gs
    mov  ax, gs
    cmp  al, 3
    jne  being_debugged
```

F.User-interface

```
    BlockInput
    NtSetInformationThread
    SuspendThread
    SwitchDesktop
```

i.BlockInput

The user32 BlockInput() function can block or unblock all mouse and keyboard events (apart from the ctrl-alt-delete key sequence). The effect remains until either the process exits or the function is called again with the opposite parameter. It is a very effective way to disable debuggers. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push 1
    call BlockInput
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    xor  ecx, ecx
    inc  ecx
    call BlockInput
```

The call requires that the calling thread has the DESKTOP_JOURNALPLAYBACK (0x0020) privilege (which is set by default). On *Windows Vista* and later, it also requires that the process is running at a high integrity level (that is, a process requires elevation if it was running in a standard or low-rights user account), if elevation is enabled and either the "HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System\EnableUIPI" registry value either does not exist (a default value of present and set is used in that case) or has a non-zero value (and which would require administrative privileges to change).

The function will not allow the input to be blocked twice in a row, nor will it allow the input to be unblocked twice in a row. Thus, if the same request is made twice to the function, then the return value should be different. This fact can be used to detect the presence of a number of tools that intercept the call, because most of them simply return success, regardless of the input. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push 1
    call BlockInput
    xchg ebx, eax
    push 1
    call BlockInput
    xor  ebx, eax
    je   being_debugged
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor  ecx, ecx
    inc  ecx
    call BlockInput
    xchg ebx, eax
    xor  ecx, ecx
    inc  ecx
    call BlockInput
    xor  ebx, eax
```

```
    je   being_debugged
```

ii.FLD

There is a problem in *OllyDbg*'s analyser of floating-point instructions, because *OllyDbg* does not disable errors during floating-point operations. This allows two values to cause floating-point errors (and thus crash *OllyDbg*) when converting from double-extended precision to integer. The problematic code is in the __fuistq() function:

```
    mov   eax, [esp+04]
    mov   edx, [esp+08]
    ...
    fld   t [edx]
    fistp q [eax]
    wait
    retn
```

The two values are +/-9.2233720368547758075e18. The problem can be demonstrated using these 32-bit codes on either the 32-bit or 64-bit versions of *Windows*:

```
    fld t [offset l1]
    ...
l1: dq  -1
    dw  403dh
```

and

```
    fld t [offset l1]
    ...
l1: dq  -1
    dw  0c03dh
```

There are several ways in which people have attempted to solve the problem, such as skipping the operation entirely, or using an alternative instruction (which exists only on a modern CPU, thereby exposing *OllyDbg* to another crash, if the instruction is not supported), all of which are wrong to varying degrees. The correct fix is simply to change the floating-point exception mask to ignore such errors. This can be achieved by loading the value 0x1333 (from the current value of 0x1332) into the control word of the FPU.

iii.NtSetInformationThread

Windows 2000 introduced a function extension which, at first glance, might appear to exist only for anti-debugging purposes. It is the ThreadHideFromDebugger (0x11) member of the ThreadInformationClass class. It can be set on a per-thread basis by calling the ntdll NtSetInformationThread() function. It is intended to be used by an external process, but any thread can use it on itself. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
push 0
push 0
push 11h ;ThreadHideFromDebugger
push -2 ;GetCurrentThread()
call NtSetInformationThread
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
xor  r9d, r9d
xor  r8d, r8d
push 11h ;ThreadHideFromDebugger
pop  rdx
push -2 ;GetCurrentThread()
pop  rcx
call NtSetInformationThread
```

When the function is called, the thread will continue to run but a debugger will no longer receive any events related to that thread. Among the missing events are that the process has terminated, if the main thread is the hidden one. The reason why the function exists is to avoid an unexpected interruption when an external process uses the ntdll RtlQueryProcessDebugInformation() function to query information about the debuggee. The ntdll RtlQueryProcessDebugInformation() function injects a thread into the debuggee in order to gather information about the process. If the injected thread is not hidden from the debugger then the debugger will gain control when the thread starts, and the debuggee will stop executing.

iv.SuspendThread

The kernel32 SuspendThread() function (or the ntdll NtSuspendThread() function) can be another very effective way to disable user-mode debuggers. This can be achieved by enumerating the threads of a given process, or searching for a named window and opening its owner thread, and then suspending that thread. The call can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows* (based on the earlier code that found the parent process and placed the process ID in the EDI register):

```
    push edi
    push 4 ;TH32CS_SNAPTHREAD
    call CreateToolhelp32Snapshot
    push 1ch ;sizeof(THREADENTRY32)
    push esp
    push eax
    xchg ebx, eax
    call Thread32First
l1: push esp
    push ebx
    call Thread32Next
    cmp  [esp+0ch], edi ;th32OwnerProcessID
    jne  l1
    push d [esp+8] ;th32ThreadID
    push 0
    push 2 ;THREAD_SUSPEND_RESUME
    call OpenThread
    push eax
    call SuspendThread
```

or using this 64-bit code on the 64-bit versions of *Windows* (based on the earlier code that found the parent process and placed the process ID in the EDI register):

```
    mov   edx, edi
    push  4 ;TH32CS_SNAPTHREAD
    pop   rcx
    call  CreateToolhelp32Snapshot
    mov   ebx, eax
    push  1ch ;sizeof(THREADENTRY32)
    pop   rbp
    enter 20h, 0
    mov   edx, ebp
    xchg  ecx, eax
    call  Thread32First
l1: mov   edx, ebp
    mov   ecx, ebx
    call  Thread32Next
    cmp   [rbp+0ch], edi ;th32OwnerProcessID
    jne   l1
    mov   r8, [rbp+8] ;th32ThreadID
    cdq
    push  2 ;THREAD_SUSPEND_RESUME
    pop   rcx
    call  OpenThread
    xchg  ecx, eax
    call  SuspendThread
```

v.SwitchDesktop

Windows NT-based platforms support multiple desktops per session. It is possible to select a different active desktop, which has the effect of hiding the windows of the previously active desktop, and with no obvious way to switch back to the old desktop (the ctrl-alt-delete key sequence will not do it). Further, the mouse and keyboard events from the debuggee's desktop will not be delivered anymore to the debugger, because their source is no longer shared. This obviously makes debugging impossible. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push eax
    ;DESKTOP_CREATEWINDOW
    ;+ DESKTOP_WRITEOBJECTS
    ;+ DESKTOP_SWITCHDESKTOP
    push 182h
    push eax
    push eax
    push eax
    push offset l1
    call CreateDesktopA
    push eax
    call SwitchDesktop
    ...
l1: db   "mydesktop", 0
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    xor  edx, edx
    push rdx
    ;DESKTOP_CREATEWINDOW
    ;+ DESKTOP_WRITEOBJECTS
    ;+ DESKTOP_SWITCHDESKTOP
    push 182h
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    mov  rcx, offset l1
    call CreateDesktopA
    xchg ecx, eax
    call SwitchDesktop
    ...
l1: db   "mydesktop", 0
```

G.Uncontrolled execution

```
    CreateProcess
    CreateThread
    DebugActiveProcess
    Enum...
    NtSetLdtEntries
```

```
QueueUserAPC
RaiseException
RtlProcessFlsData
WriteProcessMemory
Intentional exception
```

i.CreateProcess

One of the simplest ways to escape from the control of a debugger is for a process to execute another copy of itself. Typically, the process will use a synchronisation object, such as a mutex, to prevent being repeated infinitely. The first process will create the mutex, and then execute the copy of the process. The second process will not be under the debugger's control, even if the first process was. The second process will also know that it is the copy since the mutex will exist. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
xor  ebx, ebx
push offset l2
push ebx
push ebx
call CreateMutexA
call GetLastError
cmp  eax, 0b7h ;ERROR_ALREADY_EXISTS
je   l1
mov  ebp, offset l3
push ebp
call GetStartupInfoA
call GetCommandLineA
sub  esp, 10h ;sizeof(PROCESS_INFORMATION)
push esp
push ebp
push ebx
push ebx
push ebx
push ebx
push ebx
push ebx
push eax
```

```
    push ebx
    call CreateProcessA
    pop  eax
    push -1 ;INFINITE
    push eax
    call WaitForSingleObject
    call ExitProcess
l1: ...
l2: db   "mymutex", 0
l3: db   44h dup (?) ;sizeof(STARTUPINFO)
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  r8, offset l2
    xor  edx, edx
    xor  ecx, ecx
    call CreateMutexA
    call GetLastError
    cmp  eax, 0b7h ;ERROR_ALREADY_EXISTS
    je   l1
    mov  rbp, offset l3
    push rbp
    pop  rcx
    call GetStartupInfoA
    call GetCommandLineA
    mov  rsi, offset l4
    push rsi
    push rbp
    xor  ecx, ecx
    push rcx
    push rcx
    push rcx
    push rcx
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    xchg edx, eax
    call CreateProcessA
    or   rdx, -1 ;INFINITE
    mov  ecx, [rsi]
    call WaitForSingleObject
    call ExitProcess
l1: ...
```

```
l2: db   "mymutex", 0
l3: db   68h dup (?) ;sizeof(STARTUPINFO)
l4: db   18h dup (?) ;sizeof(PROCESS_INFORMATION)
```

It is quite common to see the use of the kernel32 Sleep() function, instead of the kernel32 WaitForSingleObject() function, but this introduces a race condition. The problem occurs when there is CPU-intensive activity at the time of execution. This could be because of a sufficiently complicated protection (or intentional delays) in the second process; but also actions that the user might perform while the execution is in progress, such as browsing the network or extracting files from an archive. The result is that the second process might not reach the mutex check before the delay expires; leading it to think that it is the first process. If that happens, then the process will execute yet another copy of itself. This behaviour can occur repeatedly, until one of the processes finally completes the mutex check successfully.

Note also that the first process must wait until the second process either terminates (by waiting on the process handle) or at least signals its successful start (for example, by waiting on an event handle instead of using a mutex). Otherwise, the first process might terminate before the second process performs the state check, and then the second process will think that it is the first process, and the cycle will repeat.

An extension of the self-execution method is self-debugging. Self-debugging, as the name suggests, is the act of running a copy of oneself, and then attaching to it as a debugger. It does not mean a single process debugging itself, because that is not possible. Since only one debugger can be attached to a process at a time, the second process becomes "undebuggable" by ordinary means. The first process does not even need to do anything debugger-related, though it certainly can choose to do so. In the simplest case, the first process can simply ignore any debugger-related events (such as DLL loading), except for the process termination event. In the more advanced case, the second process might contain typical anti-debugger tricks such as hard-coded breakpoints, but now they might have special meaning to the first process, making it very difficult to simulate the debugging environment using only a single process. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  ebx, ebx
    push offset l4
    push ebx
    push ebx
    call CreateMutexA
    call GetLastError
    cmp  eax, 0b7h ;ERROR_ALREADY_EXISTS
    je   l3
    mov  ebp, offset l5
    push ebp
    call GetStartupInfoA
    call GetCommandLineA
    mov  esi, offset l6
    push esi
    push ebp
    push ebx
    push ebx
    push 1 ;DEBUG_PROCESS
    push ebx
    push ebx
    push ebx
    push eax
    push ebx
    call CreateProcessA
    mov  ebx, offset l7
    jmp  l2
l1: push 10002h ;DBG_CONTINUE
    push d [esi+0ch] ;dwThreadId
    push d [esi+8] ;dwProcessId
    call ContinueDebugEvent
l2: push -1 ;INFINITE
    push ebx
    call WaitForDebugEvent
    cmp  b [ebx], 5 ;EXIT_PROCESS_DEBUG_EVENT
```

```
    jne  l1
    call ExitProcess
l3: ;execution resumes here in second process
    ...
l4: db   "mymutex", 0
l5: db   44h dup (?) ;sizeof(STARTUPINFO)
l6: db   10h dup (?) ;sizeof(PROCESS_INFORMATION)
l7: db   60h dup (?) ;sizeof(DEBUG_EVENT)
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  r8, offset l4
    xor  edx, edx
    xor  ecx, ecx
    call CreateMutexA
    call GetLastError
    cmp  eax, 0b7h ;ERROR_ALREADY_EXISTS
    je   l3
    mov  rbp, offset l5
    push rbp
    pop  rcx
    call GetStartupInfoA
    call GetCommandLineA
    mov  rsi, offset l6
    push rsi
    push rbp
    xor  ecx, ecx
    push rcx
    push rcx
    push 1 ;DEBUG_PROCESS
    push rcx
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    xchg edx, eax
    call CreateProcessA
    mov  rbx, offset l7
    jmp  l2
l1: mov  r8d, 10002h ;DBG_CONTINUE
    mov  edx, [rsi+14h] ;dwThreadId
    mov  ecx, [rsi+10h] ;dwProcessId
    call ContinueDebugEvent
l2: or   rdx, -1 ;INFINITE
```

```
    push rbx
    pop  rcx
    call WaitForDebugEvent
    cmp  b [rbx], 5 ;EXIT_PROCESS_DEBUG_EVENT
    jne  l1
    call ExitProcess
l3: ;execution resumes here in second process
    ...
l4: db   "mymutex", 0
l5: db   68h dup (?) ;sizeof(STARTUPINFO)
l6: db   18h dup (?) ;sizeof(PROCESS_INFORMATION)
l7: db   0ach dup (?) ;sizeof(DEBUG_EVENT)
```

ii.CreateThread

Threads are a simple way for the debuggee to transfer control to a memory location where execution can resume freely, unless a breakpoint is placed at the appropriate location. They can also be used to interfere with the execution of the other threads (including the main thread), and thus interfere with debugging. One example is to check periodically for software breakpoints or other memory alterations that a debugger might cause to the code stream. The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push eax
    push esp
    push eax
    push eax
    push offset l2
    push eax
    push eax
    call CreateThread
l1: ;here could be obfuscated code
    ;with lots of dummy function calls
    ;that would invite step-over
    ;and thus breakpoint insertion
    ...
```

```
l2: xor  eax, eax
    cdq
    mov  ecx, offset l2 - offset l1
    mov  esi, offset l1
l3: lodsb
    add  edx, eax ;simple sum to detect breakpoints
    loop l3
    cmp  edx, <checksum>
    jne  being_debugged
    mov  ch, 1;small delay then restart
    push ecx
    call Sleep
    jmp  l2
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor  edx, edx
    push rdx
    push rsp
    push rdx
    sub  esp, 20h
    xor  r9d, r9d
    mov  r8, offset l2
    xor  ecx, ecx
    call CreateThread
l1: ;here could be obfuscated code
    ;with lots of dummy function calls
    ;that would invite step-over
    ;and thus breakpoint insertion
    ...
l2: xor  eax, eax
    cdq
    mov  ecx, offset l2 - offset l1
    mov  rsi, offset l1
l3: lodsb
    add  edx, eax ;simple sum to detect breakpoints
    loop l3
    cmp  edx, <checksum>
    jne  being_debugged
    mov  ch, 1 ;small delay then restart
    call Sleep
    jmp  l2
```

iii.DebugActiveProcess

The kernel32 DebugActiveProcess() function (or the ntdll DbgUiDebugActiveProcess() function or the ntdll NtDebugActiveProcess() function) can be used to attach as a debugger to an already running process. Since only one debugger can be attached to a process at a time, a failure to attach to the process might indicate the presence of another debugger (though there can be other reasons for failure, such as security descriptor restrictions). The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    mov  ebp, offset l4
    push ebp
    call GetStartupInfoA
    xor  ebx, ebx
    mov  esi, offset l5
    push esi
    push ebp
    push ebx
    push ebx
    push ebx
    push ebx
    push ebx
    push ebx
    push offset l3
    push ebx
    call CreateProcessA
    push d [esi+8] ;dwProcessId
    call DebugActiveProcess
    test eax, eax
    je   being_debugged
    mov  ebx, offset l6
    jmp  l2
l1: push 10002h ;DBG_CONTINUE
    push d [esi+0ch] ;dwThreadId
    push d [esi+8] ;dwProcessId
    call ContinueDebugEvent
l2: push -1 ;INFINITE
    push ebx
    call WaitForDebugEvent
```

```
    cmp  b [ebx], 5 ;EXIT_PROCESS_DEBUG_EVENT
    jne  l1
    call ExitProcess
l3: db   "myfile", 0
l4: db   44h dup (?) ;sizeof(STARTUPINFO)
l5: db   10h dup (?) ;sizeof(PROCESS_INFORMATION)
l6: db   60h dup (?) ;sizeof(DEBUG_EVENT)
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  rbp, offset l4
    push rbp
    pop  rcx
    call GetStartupInfoA
    mov  rsi, offset l5
    push rsi
    push rbp
    xor  ecx, ecx
    push rcx
    push rcx
    push rcx
    push rcx
    sub  esp, 20h
    xor  r9d, r9d
    xor  r8d, r8d
    mov  rdx, offset l3
    call CreateProcessA
    mov  ecx, [rsi+10h] ;dwProcessId
    call DebugActiveProcess
    test eax, eax
    je   being_debugged
    mov  rbx, offset l6
    jmp  l2
l1: mov  r8d, 10002h ;DBG_CONTINUE
    mov  edx, [rsi+14h] ;dwThreadId
    mov  ecx, [rsi+10h] ;dwProcessId
    call ContinueDebugEvent
l2: or   rdx, -1 ;INFINITE
    push rbx
    pop  rcx
    call WaitForDebugEvent
    cmp  b [rbx], 5 ;EXIT_PROCESS_DEBUG_EVENT
    jne  l1
```

```
    call ExitProcess
l3: db   "myfile", 0
l4: db   68h dup (?) ;sizeof(STARTUPINFO)
l5: db   18h dup (?) ;sizeof(PROCESS_INFORMATION)
l6: db   0ach dup (?) ;sizeof(DEBUG_EVENT)
```

iv.Enum...

There are many enumeration functions, and some of them are in DLLs other than kernel32.dll. Any one of them can be used to transfer control to a memory location where execution can resume freely, unless a breakpoint is placed at the appropriate location. The call can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    push 0
    push 0
    push offset l1
    call EnumDateFormatsA
    jmp being_debugged
l1: ;execution resumes here during enumeration
    ...
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    xor  r8d, r8d
    xor  edx, edx
    mov  rcx, offset l1
    call EnumDateFormatsA
    jmp being_debugged
l1: ;execution resumes here during enumeration
    ...
```

v.GenerateConsoleCtrlEvent

When a user presses either the Ctrl-C or Ctrl-Break key combination while a console window has the focus, *Windows* checks if the event should be handled or ignored. If the event should be handled, then

Windows calls the registered console control handler, or the kernel32 CtrlRoutine() function. The kernel32 CtrlRoutine() function checks for the presence of a debugger (which is determined by reading the BeingDebugged flag in the Process Environment Block), and then issues the DBG_CONTROL_C (0x40010005) exception or the DBG_CONTROL_BREAK (0x40010008) exception, if it is. This exception can be intercepted by an exception handler or an event handler, but the exception might be consumed by the debugger instead. As a result, the missing exception can be used to infer the presence of the debugger. The application can register a Structured Exception Handler, or register an event handler by calling the kernel32 SetConsoleCtrlHandler() function. The exception can be forced to occur by calling the kernel32 GenerateConsoleCtrlEvent() function. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    ;Process Environment Block
    mov  ecx, fs:[eax+30h]
    inc  b [ecx+2]
    push eax
    push eax ;CTRL_C_EVENT
    call GenerateConsoleCtrlEvent
    jmp  $
l1: ;execution resumes here if exception occurs
    ...
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    mov  rdx, offset l1
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    push 60h
    pop  rsi
```

```
     gs:lodsq ;Process Environment Block
     inc  b [eax+2]
     push 0
     push 0 ;CTRL_C_EVENT
     call GenerateConsoleCtrlEvent
     jmp  $
 l1: ;execution resumes here if exception occurs
     ...
```

vi.NtSetInformationProcess

Perhaps because the value of the Local Descriptor Table (LDT) register is zero in user-mode in a physical (as opposed to a virtual) *Windows* environment, it is generally not supported properly (or at all) by debuggers. As a result, it can be used as a simple anti-debugger technique. Specifically, a new Local Descriptor Table entry can be created which maps to some code. This can be done on a per-process basis by calling the ntdll NtSetInformationProcess() function, and passing the ProcessLdtInformation (0x0a) member of the ProcessInformationClass class. Then, a transfer of control instruction (call, jump, ret, etc.) to the new Local Descriptor Table entry might cause the debugger to become confused about the new memory location, or even refuse to debug any further. The call can be made using this 32-bit code to examine the 32-bit versions of *Windows* (the call is not supported on the 64-bit versions of *Windows*):

```
     ;base must be <= PE->ImageBase
     ;but no need for 64kb align
     base equ 12345678h
     ;sel must have bit 2 set
     ;CPU will set bits 0 and 1
     ;even if we don't do it
     sel  equ 777h

     ;4k granular, 32-bit, present
     ;DPL3, exec-only code
     ;limit must not touch kernel mem
     ;calculate carefully to use functions
```

```
    push (base and 0ff000000h) \
       + 0c1f800h \
       + ((base shr 10h) and 0ffh)
    push (base shl 10h) + 0ffffh
    push 8
    push sel and -8 ;bits 0-2 must be clear here
    mov  eax, esp
    push 10h
    push eax
    push 0ah ;ProcessLdtInformation
    push -1 ;GetCurrentProcess()
    call NtSetInformationProcess
    ;jmp far sel:l1
    db   0eah
    dd   offset l1 - base
    dw   sel
l1: ;execution continues here but using LDT selector
    ...
```

vii.NtSetLdtEntries

The ntdll NtSetLdtEntries() function also allows setting the Local Descriptor Table values directly, but only for the current process. It is an entirely separate function with a slightly different parameter format compared to the ProcessLdtInformation technique above, but the result is the same. The call can be made using this 32-bit code to examine the 32-bit versions of *Windows* (the call is not supported on the 64-bit versions of *Windows*):

```
    ;base must be <= PE->ImageBase
    ;but no need for 64kb align
    base equ 12345678h
    ;sel must have bit 2 set
    ;CPU will set bits 0 and 1
    ;even if we don't do it
    sel  equ 777h

    xor  eax, eax
    push eax
    push eax
```

```
    push eax
    ;4k granular, 32-bit, present
    ;DPL3, exec-only code
    ;limit must not touch kernel mem
    ;calculate carefully to use functions
    push (base and 0ff000000h) \
       + 0c1f800h \
       + ((base shr 10h) and 0ffh)
    push (base shl 10h) + 0ffffh
    push sel
    call NtSetLdtEntries
    ;jmp far sel:l1
    db   0eah
    dd   offset l1 - base
    dw   sel
l1: ;execution continues here but using LDT selector
    ...
```

viii.QueueUserAPC

The kernel32 QueueUserAPC() function (or the ntdll NtQueueApcThread() function) can be used to register Asynchronous Procedure Calls (APCs). Asynchronous Procedure Calls are functions that are called when the associated thread enters an "alertable" state, such as by calling the kernel32 Sleep() function. They are also called before the thread entrypoint if they were registered before the thread began to run. They are an interesting way for the debuggee to transfer control to a memory location where execution can resume freely, unless a breakpoint is placed at the appropriate location. They can be used instead of calling the kernel32 CreateRemoteThread() function, if the target thread is known to call one of the alertable functions. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push eax
    push esp
    push eax
    push eax
```

```
     push eax ;thread entrypoint is irrelevant
     push eax
     push eax
     call CreateThread
     push eax
     push eax
     push offset l1
     call QueueUserAPC
     jmp  $
 l1: ;execution resumes here when thread starts
     ...
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
     xor  edx, edx
     push rdx
     push rsp
     push rdx
     sub  esp, 20h
     xor  r9d, r9d
     xor  r8d, r8d ;thread entrypoint is irrelevant
     xor  ecx, ecx
     call CreateThread
     xchg edx, eax
     mov  rcx, offset l1
     call QueueUserAPC
     jmp  $
 l1: ;execution resumes here when thread starts
     ...
```

ix.RaiseException

The kernel32 RaiseException() function (or the ntdll RtlRaiseException() function and the ntdll NtRaiseException() function) can be used to force certain exceptions to occur, which includes exceptions that a debugger would normally consume. Different debuggers consume different sets of exceptions. As a result, any exceptions from the appropriate set, when raised in the presence of the particular debugger, will be delivered to the debugger instead of the debuggee. The missing exception can be used to infer the presence of the particular debugger. Some debuggers allow specific (or all) exceptions to be delivered to the debuggee in all circumstances. However, this can result in the debugger losing control over the debuggee. The exception that is consumed most commonly is the DBG_RIPEVENT exception (0x40010007). The check can be made using this 32-bit code to examine the 32-bit *Windows* environment on either the 32-bit or 64-bit versions of *Windows*:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    push eax
    push eax
    push eax
    push 40010007h ;DBG_RIPEVENT
    call RaiseException
    jmp  being_debugged
l1: ;execution resumes here due to execption
    ...
```

or using this 64-bit code to examine the 64-bit *Windows* environment:

```
    mov  rdx, offset l1
    xor  ecx, ecx
    inc  ecx
    call AddVectoredExceptionHandler
    xor  r9d, r9d
    xor  r8d, r8d
    cdq
    mov  ecx, 40010007h ;DBG_RIPEVENT
    call RaiseException
    jmp  being_debugged
l1: ;execution resumes here due to exception
    ...
```

One debugger is known to allow being directed to perform specific actions based on the parameters that are supplied along with the exception, such as replacing a software breakpoint with an arbitrary byte anywhere in the process memory.

One debugger is known to crash when the function is called directory, due to an off-by-one bug when attempting to print some related logging information.

x.RtlProcessFlsData

The ntdll RtlProcessFlsData() function is an undocumented function that was introduced in *Windows Vista*. It is called by the kernel32 FlsSetValue() and kernel32 DeleteFiber() functions. When called with the appropriate parameter and memory values, the function will execute code at a user-specified pointer in memory. If a debugger is unaware of this fact, then execution control might be lost. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push 30h
    pop  eax
    mov  ecx, fs:[eax]
    mov  ah, 2
    inc  d [ecx+eax-4] ;must be at least 1
    mov  esi, offset l2-4
    mov  [ecx+eax-24h], esi
    lodsd
    push esi
    call RtlProcessFlsData
    jmp  being_debugged
l1: ;execution resumes here during processing
    ...
l2: dd   0, offset l1, 0, 1
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    push 60h
    pop  rax
    mov  rcx, gs:[rax]
    mov  ah, 3
```

```
    inc  d [rcx+rax-10h] ;must be at least 1
    mov  rsi, offset l2-8
    mov  [rcx+rax-40h], rsi
    lodsq
    push rsi
    pop  rcx
    call RtlProcessFlsData
    jmp  being_debugged
l1: ;execution resumes here during processing
    ...
l2: dq   0, offset l1, 0, 1
```

xi.WriteProcessMemory

The kernel32 WriteProcessMemory() function (or the ntdll NtWriteVirtualMemory() function) can be used in much the same way as the kernel32 ReadFile() function, as described above, except that the source of the data is the process memory space instead of the file on disk. It can be used on the current process. The call can be made using this 32-bit code on either the 32-bit or 64-bit versions of *Windows*:

```
    push 0
    ;replace byte at l1 with byte at l2
    ;step-over will also result
    ;in uncontrolled execution
    push 1
    push offset l2
    push offset l1
    push -1 ;GetCurrentProcess()
    call WriteProcessMemory
l1: int  3
l2: nop
```

or using this 64-bit code on the 64-bit versions of *Windows*:

```
    push 0
    sub  esp, 20h
    ;replace byte at l1 with byte at l2
    ;step-over will also result
```

```
    ;in uncontrolled execution
    push 1
    pop  r9
    mov  r8, offset l2
    mov  rdx, offset l1
    or   rcx, -1 ;GetCurrentProcess()
    call WriteProcessMemory
l1: int  3
l2: nop
```

One way to defeat this technique is to use hardware breakpoints instead of software breakpoints when stepping over function calls.

xii.Intentional exceptions

An exception that is not handled by a debugger is an easy way for the debuggee to transfer control to a memory location where execution can resume freely, unless a breakpoint is placed at the appropriate location. A hint that such an attempt will be made often looks like this 32-bit code:

```
    xor  eax, eax
    push offset handler ;step 1
    push d fs:[eax] ;step 2
    mov  fs:[eax], esp ;step 3
    [code to force exception]
```

This is the Structured Exception Handling method. It exists only in 32-bit *Windows* environments. In 64-bit *Windows* environments, Vectored Exception Handling is used instead for registering an exception handler dynamically. At some point after registering an exception handler, an exception is generated. There are many ways to generate an exception, such as by using illegal or privileged instructions, or illegal memory accesses.

There are ways to obfuscate some of the instructions in the sequence. The first one is to remove the references to the FS selector. The value at fs:[18h] is a pointer to the FS region, but accessible using any selector other than GS, in this way:

```
mov  eax, fs:[18h]
push d [eax] ;step 2
mov  [eax], esp ;step 3
```

Then there is a way to make the push indirect:

```
mov   ebp, [eax]
enter 0, 0 ;step 2
mov   [eax], ebp ;step 3
```

There are other ways to get the base address for the FS selector, such as by using the kernel32 GetThreadSelectorEntry() function, but the general sequence of steps remains the same. However, there is one published method[4] that makes even the memory write indirect. It looks like this:

```
    push 8
    pop  eax
    call l2
    push -1 ;push fs:[0] ;step 2
    mov  ecx, fs
    mov  ss, ecx
    xchg esp, eax
    call l1 ;change stack limit
l1: push eax ;mov fs:[0], esp ;step 3
    mov  esp, ebp ;point esp somewhere legal
l2: pop  esp
    call esp ;push offset handler ;step 1
    ;handler code here
```

This code assumes that there is no writable memory before the start. If there were, then the code could be shortened by two bytes. It also assumes that the stack is lower in memory than the code is. If this is not the case, then when the exception occurs, *Windows* will terminate the process. Interestingly, this version works only in the 32-bit

[4] http://vx.netlux.org/lib/vrg03.html

environment on the 64-bit versions of *Windows*, due to an assumption about the behaviour of selectors when an exception occurs. Specifically, the location l2 is reached twice, the second time with the SS selector value equal to the FS selector value. The assumption is that the "pop esp" instruction at l2 will cause an exception at that time (because the ESP register value will be beyond the limits of the FS segment), and that the SS selector value will be restored to its correct value. However, this restoration occurs only on the 64-bit versions of *Windows*. It does not occur on the 32-bit versions of *Windows*. To achieve compatibility with 32-bit versions of Windows, and correct all of the assumptions at the same time, the code would need to look like this:

```
    ;writable page before this point
    push 0ch
    pop  eax
    call l2
    push -1 ;push fs:[0] ;step 2
    push fs
    pop  ss
    xchg esp, eax
    push esp ;change stack base
    call l1 ;change stack limit
l1: push eax ;mov fs:[0], esp ;step 3
    mov  ecx, ds
    mov  ss, ecx ;restore ss
    xchg esp, eax ;point esp somewhere legal
    push esp ;point to exception instruction
l2: pop  esp ;get address so call will fault
    call esp ;push offset handler ;step 1
    ;handler code here
```

a.Nanomites

Nanomites typically work by replacing branch instructions with an "int 3" instruction, and then using tables in the unpacking code to determine if the "int 3" is a nanomite or a debug break. If the "int 3" is a nanomite, then the tables will also be used to determine whether or not the branch should be taken, the address of the destination, if the branch is taken, and how large the instruction is, if the branch is not taken.

A process that is protected by nanomites typically requires self-debugging. This allows the debugger to process the exceptions that are generated by the debuggee when the nanomite is hit.

However, there is no requirement for self-debugging, except as an anti-debugging measure. A single process could register its own exception handler and process the exceptions on its own. There is at least one virus which is known to exhibit the most extreme version of this behaviour. The virus orders every one of its instructions randomly, and links them all using nanomites that carry information about the location of the next instruction.

H.Conclusion

As we can see, there are very many anti-debugging tricks. Some are known and some are new, and not all of them have good defences. What we need is the ultimate debugger that knows them all.