Anatomy of the Process Environment Block (PEB) (Windows Internals)

ntopcode.wordpress.com/2018/02/26/anatomy-of-the-process-environment-block-peb-windows-internals/

February 26, 2018

The Process Environment Block (PEB) is a wonderful thing, and I'd be lying if I told you that I didn't love it. It has been present in Windows since the introduction of the Win2k (Windows 2000) and it has been improved through newer versions of Windows ever since. On earlier versions of Windows, it could be abused to do some nasty things like hiding loaded modules present within a process (to prevent them from being found – obviously this is not a beautiful thing though).

What is this magic so-called "Process Environment (PEB)"? The PEB is a structure which holds data about the current process under it's field values – some fields being structures themselves to hold even more data. Every process has it's own PEB and the Windows Kernel will also have access to the PEB of every user-mode process so it can keep track of certain data stored within it.

Where does this sorcery come from? The PEB structure comes from the Windows Kernel (although is accessible in user-mode as well). The PEB comes from the Thread Environment Block (TEB) which also happens to be commonly referred to as the Thread Information Block (TIB). The TEB is responsible for holding data about the current thread – every thread has it's own TEB structure.

Can the Thread Environment Block or the Process Environment Block be abused for malicious purposes? Of course they can! In fact, they ***have*** been abused for malicious purposes in the past but Microsoft has made many changes over the recent years to help prevent this. An example would be in the past where rootkits would inject a DLL into another running process, and then access the PEB structure of the current process they had injected into (the PPEB structure is a pointer to the PEB structure) so they could locate the list of loaded modules and remove their own module from the list… Thus hiding their injected module from view when someone enumerates the loaded modules of the affected process. This is known as memory patching because you would be modifying memory by patching the PEB. Microsoft's mitigation for this behavior was to prevent the manual altering of the list which represents the loaded modules in user-mode – you can still access it for reading the data in user-mode though and you can still patch the memory from kernel-mode.

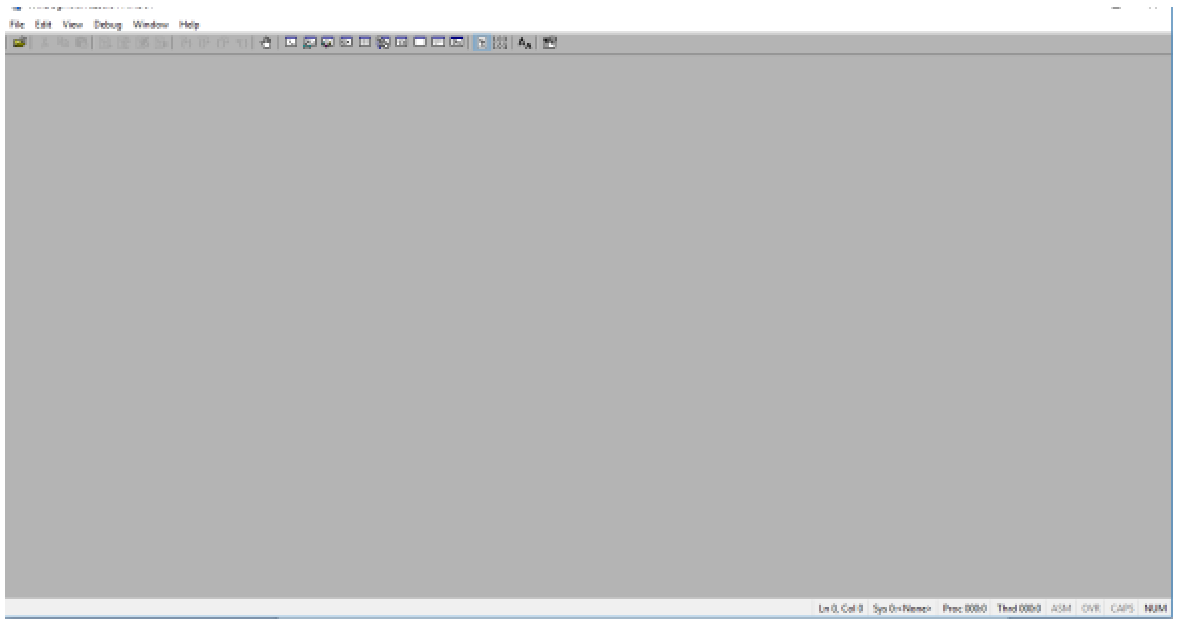
This article will be split up into two different sections: theory and user-mode practical.

Theoretical

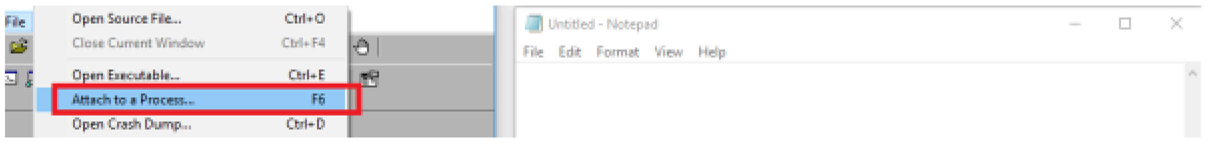
We're going to take a look at the Thread Environment Block (TEB) structure using WinDbg. Since the TEB structure is available in user-mode, and used by user-mode Windows components such as NTDLL and KERNEL32, we won't require kernel-debugging to query about the structure.

Bear in mind that you will need to have your symbols correctly setup otherwise you will fail with the next upcoming steps, please see the following URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).aspx

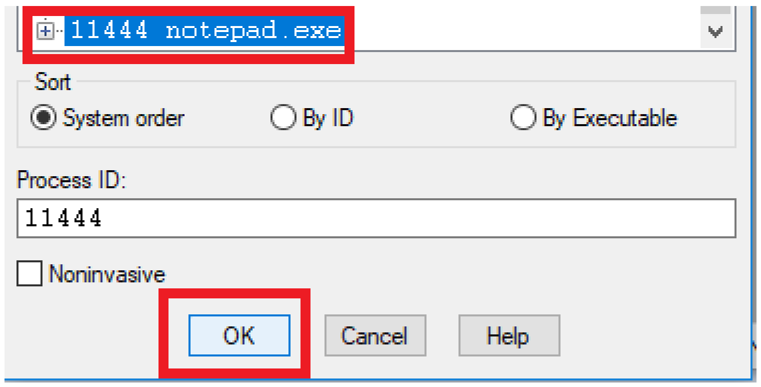
We'll start by opening up WinDbg – I'll be opening up the 64-bit version.


File Edit View Debug Window Help
Ln 0, Col 0 Sys 0:<None> Proc 000:0 Thrd 000:0 ASM OVR CAPS NUM

WinDbg default view.
Now we'll open up notepad.exe. Once it is open, we can attach to notepad.exe in WinDbg by going to **File -> Attach to a Process -> notepad.exe**. Alternatively, you can use the default hot-key which should be F6.


File
Open Source File... Ctrl+O
Close Current Window Ctrl+F4
Open Executable... Ctrl+E
Attach to a Process... F6
Open Crash Dump... Ctrl+D
Untitled - Notepad
File Edit Format View Help

Attaching to a process via WinDbg. 1/2

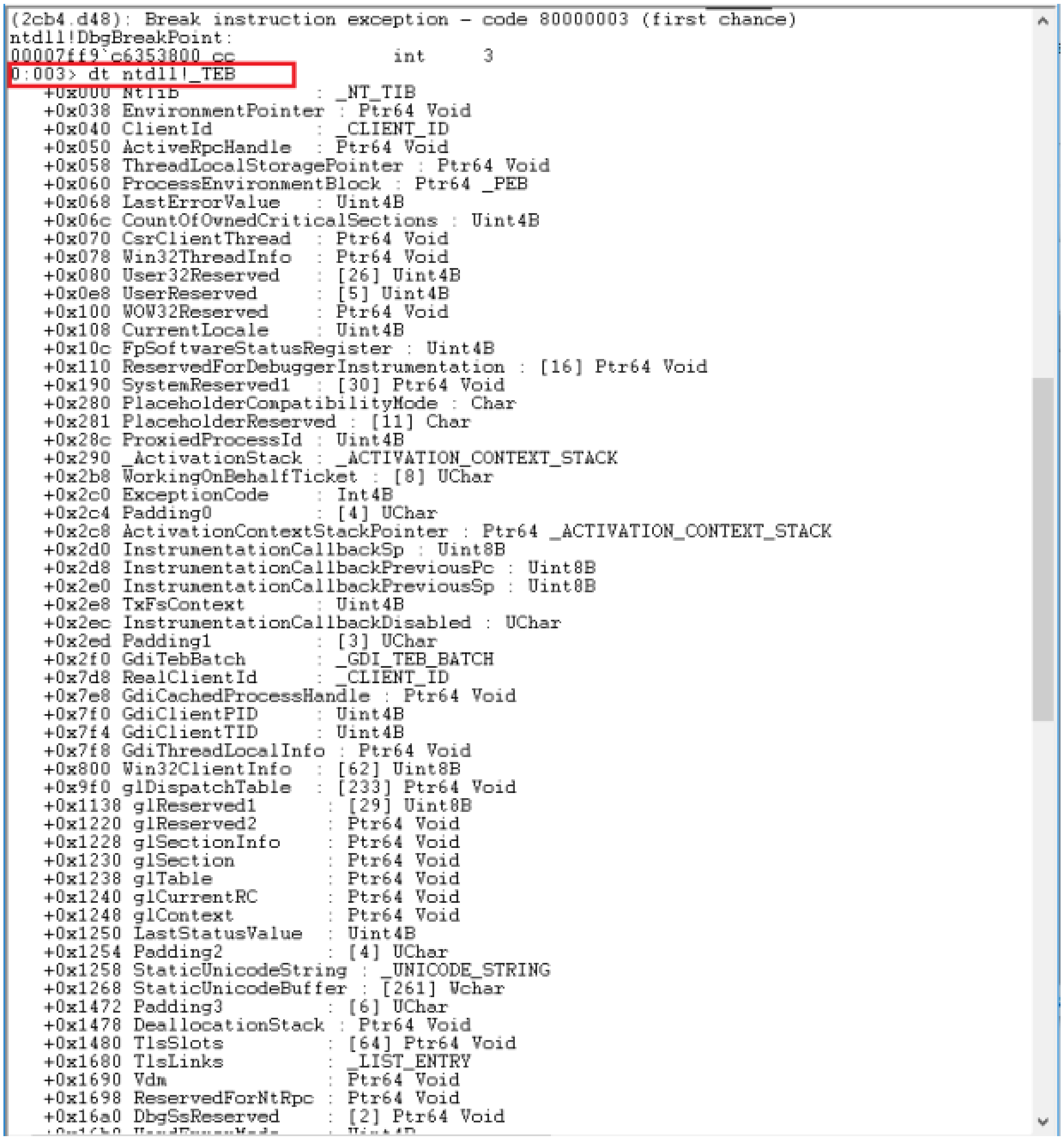


Attaching to a process via WinDbg. 2/2
After doing this, the WinDbg *command window* will be displayed. The command window is the work-space we will have to enter commands at our own discretion to get back various desired results. For example, if we wish to manipulate something, or query information about something, we can do this with a command. WinDbg has a whole wide-range of commands available and you can learn more about that here: http://windbg.info/doc/1-common-cmds.html

We'll be using the *dt* instruction. "dt" stands for "Display Type" and can be used to display information about a specific data-type, including structures. In our case, it is more than appropriate because it supports structures and we need to find out information about the TEB structure.

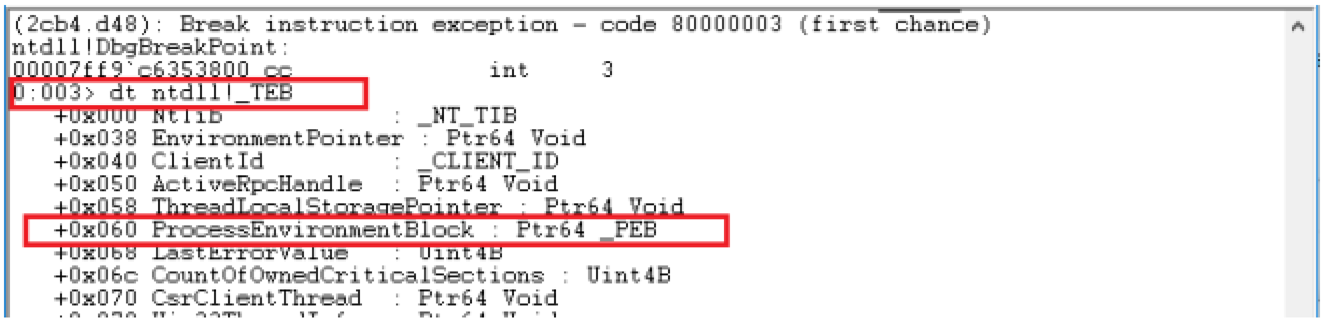
We can use the following instruction to query information about the TEB structure.

```
dt ntdll!_TEB
```



```
(2cb4.d48): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff9`c6353800 cc              int     3
0:003> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : Ptr64 Void
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : Ptr64 Void
   +0x058 ThreadLocalStoragePointer : Ptr64 Void
   +0x060 ProcessEnvironmentBlock : Ptr64 _PEB
   +0x068 LastErrorValue   : Uint4B
   +0x06c CountOfOwnedCriticalSections : Uint4B
   +0x070 CsrClientThread  : Ptr64 Void
   +0x078 Win32ThreadInfo  : Ptr64 Void
   +0x080 User32Reserved   : [26] Uint4B
   +0x0e8 UserReserved     : [5] Uint4B
   +0x100 WOW32Reserved    : Ptr64 Void
   +0x108 CurrentLocale    : Uint4B
   +0x10c FpSoftwareStatusRegister : Uint4B
   +0x110 ReservedForDebuggerInstrumentation : [16] Ptr64 Void
   +0x190 SystemReserved1  : [30] Ptr64 Void
   +0x280 PlaceholderCompatibilityMode : Char
   +0x281 PlaceholderReserved : [11] Char
   +0x28c ProxiedProcessId : Uint4B
   +0x290 _ActivationStack : _ACTIVATION_CONTEXT_STACK
   +0x2b8 WorkingOnBehalfTicket : [8] UChar
   +0x2c0 ExceptionCode    : Int4B
   +0x2c4 Padding0         : [4] UChar
   +0x2c8 ActivationContextStackPointer : Ptr64 _ACTIVATION_CONTEXT_STACK
   +0x2d0 InstrumentationCallbackSp : Uint8B
   +0x2d8 InstrumentationCallbackPreviousPc : Uint8B
   +0x2e0 InstrumentationCallbackPreviousSp : Uint8B
   +0x2e8 TxFsContext      : Uint4B
   +0x2ec InstrumentationCallbackDisabled : UChar
   +0x2ed Padding1         : [3] UChar
   +0x2f0 GdiTebBatch      : _GDI_TEB_BATCH
   +0x7d8 RealClientId     : _CLIENT_ID
   +0x7e8 GdiCachedProcessHandle : Ptr64 Void
   +0x7f0 GdiClientPID     : Uint4B
   +0x7f4 GdiClientTID     : Uint4B
   +0x7f8 GdiThreadLocalInfo : Ptr64 Void
   +0x800 Win32ClientInfo  : [62] Uint8B
   +0x9f0 glDispatchTable  : [233] Ptr64 Void
   +0x1138 glReserved1      : [29] Uint8B
   +0x1220 glReserved2      : Ptr64 Void
   +0x1228 glSectionInfo    : Ptr64 Void
   +0x1230 glSection        : Ptr64 Void
   +0x1238 glTable          : Ptr64 Void
   +0x1240 glCurrentRC      : Ptr64 Void
   +0x1248 glContext        : Ptr64 Void
   +0x1250 LastStatusValue  : Uint4B
   +0x1254 Padding2         : [4] UChar
   +0x1258 StaticUnicodeString : _UNICODE_STRING
   +0x1268 StaticUnicodeBuffer : [261] Wchar
   +0x1472 Padding3         : [6] UChar
   +0x1478 DeallocationStack : Ptr64 Void
   +0x1480 TlsSlots         : [64] Ptr64 Void
   +0x1680 TlsLinks         : _LIST_ENTRY
   +0x1690 Vdm              : Ptr64 Void
   +0x1698 ReservedForNtRpc : Ptr64 Void
   +0x16a0 DbgSsReserved    : [2] Ptr64 Void
```

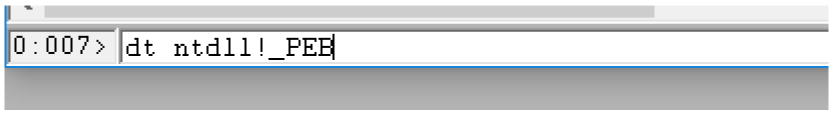
WinDbg command (dt) for the _TEB structure.
We can see already that there are many fields of the structure, so many fields that they all don't fit on the singular image view. However, if we look towards the very top of the structure, we'll find the Process Environment Block's field.

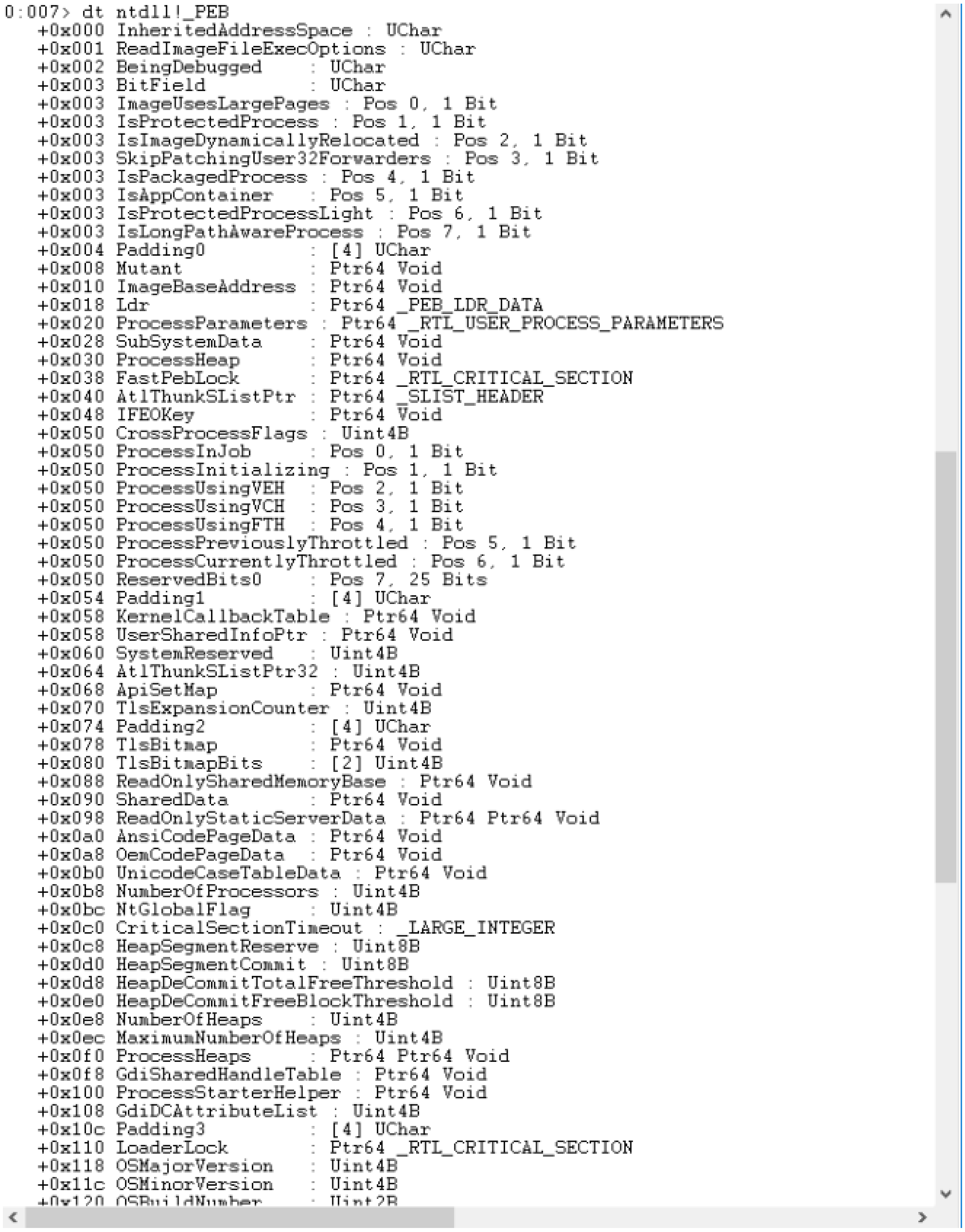

```
(2cb4.d48): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ff9`c6353800 cc              int     3
0:003> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : Ptr64 Void
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : Ptr64 Void
   +0x058 ThreadLocalStoragePointer : Ptr64 Void
   +0x060 ProcessEnvironmentBlock : Ptr64 _PEB
   +0x068 LastErrorValue   : Uint4B
   +0x06c CountOfOwnedCriticalSections : Uint4B
   +0x070 CsrClientThread  : Ptr64 Void
```

Highlighting the ProcessEnvironmentBlock field of the _TEB structure.
We can see that WinDbg is labelling the data-type for the field as “Ptr64 _PEB”. This simply means that the data-type is a pointer to the PEB structure (PPEB). Since we are debugging a 64-bit compiled program (notepad.exe since our OS architecture is 64-bit), the addresses are 8 bytes instead of 4 bytes like on a 32-bit environment, which is why “64” is appended to the “Ptr”.

We can view the fields of the PEB structure with the following WinDbg command.

```
dt ntdll!_PEB
```


```
0:007> dt ntdll!_PEB
```



```
0:007> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer   : Pos 5, 1 Bit
   +0x003 IsProtectedProcessLight : Pos 6, 1 Bit
   +0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
   +0x004 Padding0         : [4] UChar
   +0x008 Mutant           : Ptr64 Void
   +0x010 ImageBaseAddress : Ptr64 Void
   +0x018 Ldr              : Ptr64 _PEB_LDR_DATA
   +0x020 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : Ptr64 Void
   +0x030 ProcessHeap      : Ptr64 Void
   +0x038 FastPebLock      : Ptr64 _RTL_CRITICAL_SECTION
   +0x040 AtlThunkSListPtr : Ptr64 _SLIST_HEADER
   +0x048 IFEOKey          : Ptr64 Void
   +0x050 CrossProcessFlags : Uint4B
   +0x050 ProcessInJob     : Pos 0, 1 Bit
   +0x050 ProcessInitializing : Pos 1, 1 Bit
   +0x050 ProcessUsingVEH  : Pos 2, 1 Bit
   +0x050 ProcessUsingVCH  : Pos 3, 1 Bit
   +0x050 ProcessUsingFTH  : Pos 4, 1 Bit
   +0x050 ProcessPreviouslyThrottled : Pos 5, 1 Bit
   +0x050 ProcessCurrentlyThrottled : Pos 6, 1 Bit
   +0x050 ReservedBits0    : Pos 7, 25 Bits
   +0x054 Padding1         : [4] UChar
   +0x058 KernelCallbackTable : Ptr64 Void
   +0x058 UserSharedInfoPtr : Ptr64 Void
   +0x060 SystemReserved   : Uint4B
   +0x064 AtlThunkSListPtr32 : Uint4B
   +0x068 ApiSetMap        : Ptr64 Void
   +0x070 TlsExpansionCounter : Uint4B
   +0x074 Padding2         : [4] UChar
   +0x078 TlsBitmap        : Ptr64 Void
   +0x080 TlsBitmapBits    : [2] Uint4B
   +0x088 ReadOnlySharedMemoryBase : Ptr64 Void
   +0x090 SharedData       : Ptr64 Void
   +0x098 ReadOnlyStaticServerData : Ptr64 Ptr64 Void
   +0x0a0 AnsiCodePageData : Ptr64 Void
   +0x0a8 OemCodePageData  : Ptr64 Void
   +0x0b0 UnicodeCaseTableData : Ptr64 Void
   +0x0b8 NumberOfProcessors : Uint4B
   +0x0bc NtGlobalFlag     : Uint4B
   +0x0c0 CriticalSectionTimeout : _LARGE_INTEGER
   +0x0c8 HeapSegmentReserve : Uint8B
   +0x0d0 HeapSegmentCommit : Uint8B
   +0x0d8 HeapDeCommitTotalFreeThreshold : Uint8B
   +0x0e0 HeapDeCommitFreeBlockThreshold : Uint8B
   +0x0e8 NumberOfHeaps    : Uint4B
   +0x0ec MaximumNumberOfHeaps : Uint4B
   +0x0f0 ProcessHeaps     : Ptr64 Ptr64 Void
   +0x0f8 GdiSharedHandleTable : Ptr64 Void
   +0x100 ProcessStarterHelper : Ptr64 Void
   +0x108 GdiDCAttributeList : Uint4B
   +0x10c Padding3         : [4] UChar
   +0x110 LoaderLock       : Ptr64 _RTL_CRITICAL_SECTION
   +0x118 OSMajorVersion   : Uint4B
   +0x11c OSMinorVersion   : Uint4B
   +0x120 OSBuildNumber    : Uint2B
```

WinDbg command (dt) for the _PEB structure.
The WinDbg output is below.

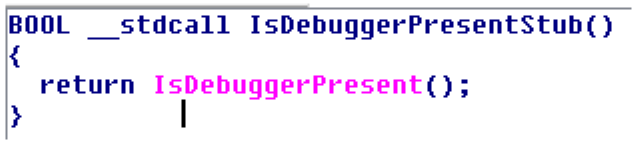
```
0:007> dt ntdll!_PEB
 +0x000 InheritedAddressSpace : UChar
 +0x001 ReadImageFileExecOptions : UChar
 +0x002 BeingDebugged : UChar
 +0x003 BitField : UChar
 +0x003 ImageUsesLargePages : Pos 0, 1 Bit
 +0x003 IsProtectedProcess : Pos 1, 1 Bit
 +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
 +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
 +0x003 IsPackagedProcess : Pos 4, 1 Bit
 +0x003 IsAppContainer : Pos 5, 1 Bit
 +0x003 IsProtectedProcessLight : Pos 6, 1 Bit
 +0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
 +0x004 Padding0 : [4] UChar
 +0x008 Mutant : Ptr64 Void
 +0x010 ImageBaseAddress : Ptr64 Void
 +0x018 Ldr : Ptr64 _PEB_LDR_DATA
 +0x020 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS
 +0x028 SubSystemData : Ptr64 Void
 +0x030 ProcessHeap : Ptr64 Void
 +0x038 FastPebLock : Ptr64 _RTL_CRITICAL_SECTION
 +0x040 AtlThunkSListPtr : Ptr64 _SLIST_HEADER
 +0x048 IFEOKey : Ptr64 Void
 +0x050 CrossProcessFlags : Uint4B
 +0x050 ProcessInJob : Pos 0, 1 Bit
 +0x050 ProcessInitializing : Pos 1, 1 Bit
 +0x050 ProcessUsingVEH : Pos 2, 1 Bit
 +0x050 ProcessUsingVCH : Pos 3, 1 Bit
 +0x050 ProcessUsingFTH : Pos 4, 1 Bit
 +0x050 ProcessPreviouslyThrottled : Pos 5, 1 Bit
 +0x050 ProcessCurrentlyThrottled : Pos 6, 1 Bit
 +0x050 ReservedBits0 : Pos 7, 25 Bits
 +0x054 Padding1 : [4] UChar
 +0x058 KernelCallbackTable : Ptr64 Void
 +0x058 UserSharedInfoPtr : Ptr64 Void
 +0x060 SystemReserved : Uint4B
 +0x064 AtlThunkSListPtr32 : Uint4B
 +0x068 ApiSetMap : Ptr64 Void
 +0x070 TlsExpansionCounter : Uint4B
 +0x074 Padding2 : [4] UChar
 +0x078 TlsBitmap : Ptr64 Void
 +0x080 TlsBitmapBits : [2] Uint4B
 +0x088 ReadOnlySharedMemoryBase : Ptr64 Void
 +0x090 SharedData : Ptr64 Void
 +0x098 ReadOnlyStaticServerData : Ptr64 Ptr64 Void
 +0x0a0 AnsiCodePageData : Ptr64 Void
 +0x0a8 OemCodePageData : Ptr64 Void
 +0x0b0 UnicodeCaseTableData : Ptr64 Void
 +0x0b8 NumberOfProcessors : Uint4B
 +0x0bc NtGlobalFlag : Uint4B
 +0x0c0 CriticalSectionTimeout : _LARGE_INTEGER
 +0x0c8 HeapSegmentReserve : Uint8B
 +0x0d0 HeapSegmentCommit : Uint8B
 +0x0d8 HeapDeCommitTotalFreeThreshold : Uint8B
 +0x0e0 HeapDeCommitFreeBlockThreshold : Uint8B
```

```
+0x0e8 NumberOfHeaps : Uint4B
+0x0ec MaximumNumberOfHeaps : Uint4B
+0x0f0 ProcessHeaps : Ptr64 Ptr64 Void
+0x0f8 GdiSharedHandleTable : Ptr64 Void
+0x100 ProcessStarterHelper : Ptr64 Void
+0x108 GdiDCAttributeList : Uint4B
+0x10c Padding3 : [4] UChar
+0x110 LoaderLock : Ptr64 _RTL_CRITICAL_SECTION
+0x118 OSMajorVersion : Uint4B
+0x11c OSMinorVersion : Uint4B
+0x120 OSBuildNumber : Uint2B
+0x122 OSCSDVersion : Uint2B
+0x124 OSPlatformId : Uint4B
+0x128 ImageSubsystem : Uint4B
+0x12c ImageSubsystemMajorVersion : Uint4B
+0x130 ImageSubsystemMinorVersion : Uint4B
+0x134 Padding4 : [4] UChar
+0x138 ActiveProcessAffinityMask : Uint8B
+0x140 GdiHandleBuffer : [60] Uint4B
+0x230 PostProcessInitRoutine : Ptr64 void
+0x238 TlsExpansionBitmap : Ptr64 Void
+0x240 TlsExpansionBitmapBits : [32] Uint4B
+0x2c0 SessionId : Uint4B
+0x2c4 Padding5 : [4] UChar
+0x2c8 AppCompatFlags : _ULARGE_INTEGER
+0x2d0 AppCompatFlagsUser : _ULARGE_INTEGER
+0x2d8 pShimData : Ptr64 Void
+0x2e0 AppCompatInfo : Ptr64 Void
+0x2e8 CSDVersion : _UNICODE_STRING
+0x2f8 ActivationContextData : Ptr64 _ACTIVATION_CONTEXT_DATA
+0x300 ProcessAssemblyStorageMap : Ptr64 _ASSEMBLY_STORAGE_MAP
+0x308 SystemDefaultActivationContextData : Ptr64 _ACTIVATION_CONTEXT_DATA
+0x310 SystemAssemblyStorageMap : Ptr64 _ASSEMBLY_STORAGE_MAP
+0x318 MinimumStackCommit : Uint8B
+0x320 FlsCallback : Ptr64 _FLS_CALLBACK_INFO
+0x328 FlsListHead : _LIST_ENTRY
+0x338 FlsBitmap : Ptr64 Void
+0x340 FlsBitmapBits : [4] Uint4B
+0x350 FlsHighIndex : Uint4B
+0x358 WerRegistrationData : Ptr64 Void
+0x360 WerShipAssertPtr : Ptr64 Void
+0x368 pUnused : Ptr64 Void
+0x370 pImageHeaderHash : Ptr64 Void
+0x378 TracingFlags : Uint4B
+0x378 HeapTracingEnabled : Pos 0, 1 Bit
+0x378 CritSecTracingEnabled : Pos 1, 1 Bit
+0x378 LibLoaderTracingEnabled : Pos 2, 1 Bit
+0x378 SpareTracingBits : Pos 3, 29 Bits
+0x37c Padding6 : [4] UChar
+0x380 CsrServerReadOnlySharedMemoryBase : Uint8B
+0x388 TppWorkerpListLock : Uint8B
+0x390 TppWorkerpList : _LIST_ENTRY
+0x3a0 WaitOnAddressHashTable : [128] Ptr64 Void
+0x7a0 TelemetryCoverageHeader : Ptr64 Void
+0x7a8 CloudFileFlags : Uint4B
```

As we can see, there's a lot of fields for the PEB structure. We'll only be focusing on a select few of them during the practical sections though.

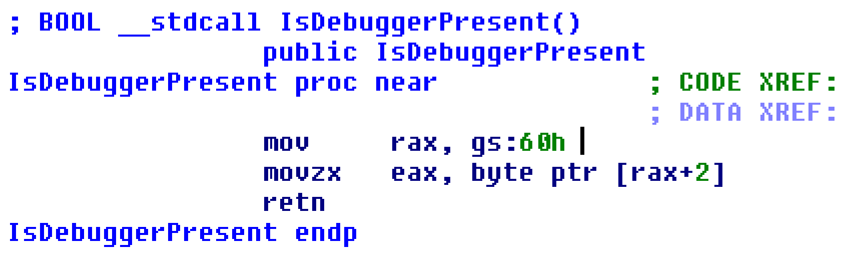
Before we can continue, we need to briefly talk about how the Process Environment Block is actually found. It's located at FS:[0x30] in the Thread Environment Block/Thread Information Block for 32-bit processes, and it's located at GS:[0x60] for 64-bit processes.

To start off, the third field of the PEB structure ("BeingDebugged") can be read to determine if the current process is attached to via a debugger – this is one vector which is commonly closed by analysts who are debugging malicious software, because malicious software tends to keep a close-eye out for debuggers and other analysis tools to make things more difficult for malware analysts. There's a routine from the Win32 API called IsDebuggerPresent (KERNEL32) and the routine works by checking the BeingDebugged field of the PEB structure. We can validate this by reverse-engineering kernel32.dll ourselves.


```
BOOL __stdcall IsDebuggerPresentStub()
{
  return IsDebuggerPresent();
}
```

IDA pseudo-code for IsDebuggerPresentStub (KERNEL32 – Windows 8+).
As we can see, kernel32.dll has a routine named IsDebuggerPresentStub which calls IsDebuggerPresent. This is because the environment I'm getting these images from is Windows 10 64-bit, and Microsoft moved to using KernelBase.dll (introduced starting Windows 8). However, for backwards-compatibility, kernel32.dll is still pushed for usage by their documentation – and if they had dropped support for it then they would have to have moved more than they have across to a new module project, and there'd have been a lot of incompatible software for Windows 8+ at the time.

Therefore, we need to take a look at KernelBase.dll.


```
; BOOL __stdcall IsDebuggerPresent()
                public IsDebuggerPresent
IsDebuggerPresent proc near             ; CODE XREF:
                                        ; DATA XREF:
                mov     rax, gs:60h
                movzx   eax, byte ptr [rax+2]
                retn
IsDebuggerPresent endp
```

Disassembly for IsDebuggerPresent (KERNEL32 / KERNELBASE).
Perfect! KernelBase.dll has an exported routine named IsDebuggerPresent. We're going to debunk what the above disassembly is telling us.

1. The address of the Process Environment Block is being moved into the RAX register. Since we're looking at the 64-bit compiled version of KernelBase.dll, 64-bit registers are being used. The Process Environment Block is located at + 0x60 for 64-bit processes.
2. The value from the BeingDebugged field under the Process Environment Block is being extracted and put into the EAX register. The data-type for the BeingDebugged field is UCHAR (which is one byte), and it's offset is 0x002 – the first field of the PEB structure is located at 0x000 which means the third field (which is the BeingDebugged field) is located +2 bytes from this address. Since the RAX register is holding the address to the Process Environment Block, (RAX + 2) is performed to reach the address of the BeingDebugged field.
3. Returning with the RETN instruction. Since the value for the BeingDebugged field of the PEB structure is held within the EAX register, the caller of the routine is going to return the value stored within the BeingDebugged field.

A routine like IsDebuggerPresent (KERNEL32 / KERNELBASE) might be an obvious sign for a malware analyst who is taking a look at the API calls being made by a sample therefore some malware samples will manually access the PEB structure to check – doing this is stealthier and usually less-expected.

The next fields we're going to briefly talk about are the IsProtectedProcess and IsProtectedProcessLight fields of the Process Environment Block.

These fields can be used to determine if the current process is "protected" or not, hence the "ProtectedProcess" key-word in the field names. In Windows, there's multiple process protection mechanisms although the former (non-Light variant) has been around a lot longer than the Process Protection Light (PPL) variant. Standard process protection mechanism in Windows has been around since Windows Vista, however the PPL feature came into play starting Windows 8. Microsoft use these mechanisms to protect their own System processes from being abused by malicious software or forcefully shut-down by a third-party source (because for some Windows processes this can cause the system to bug-check/improperly function). If we can access these fields within the Process Environment Block, then we can check if the current process is protected or not by Windows. All of this is enforced from kernel-mode by the Windows Kernel using the undocumented and opaque EPROCESS structure, and you cannot write to these fields in the PEB structure and have the changes take effect because it won't update the EPROCESS structure for the current process.

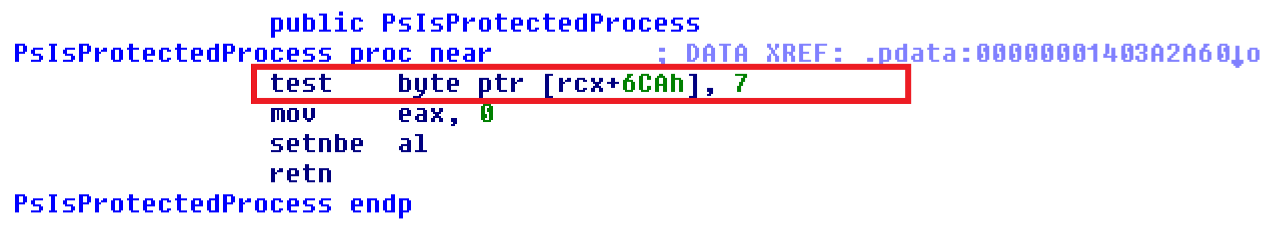
The standard process protection mechanism is used by Windows system processes. This mechanism is enforced from within the Windows Kernel and it's not supposed to be used by third-parties, and it helps prevent system processes from being exploited by attackers (or forcefully shut-down – the Operating System cannot function properly without it's critical user-mode components). On top of this, Windows will set the state of various system processes to "critical", and this is flag-based and will cause the system to be forcefully crashed (via a bug-check) if the "critical" processes become terminated. There are two different implementations for the “critical” state: critical processes and critical threads. Setting a process as critical will cause the bug-check once the process has been terminated, and setting a thread as critical will cause the bug-check once the thread has been terminated. Usually, the former is more appropriate because threads come and go regularly (e.g. spawn a new thread to handle an operation simultaneously and then the thread will be terminated once it returns back it’s status from the operation). Windows does not set “threads” as critical as far as I am aware, although it will set specific processes as critical (processes like csrss.exe).

We’re going to take a look at how the process protection mechanism which is built-into Windows actually works very briefly using Interactive Disassembler and WinDbg.

We can easily check using the following routines.

1. PsIsProtectedProcess (NTOSKRNL)
2. PsIsProtectedProcessLight (NTOSKRNL)

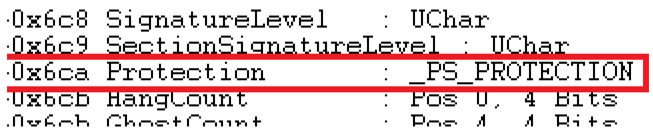
Both of the above routines are undocumented but they are still exported by the Windows Kernel.



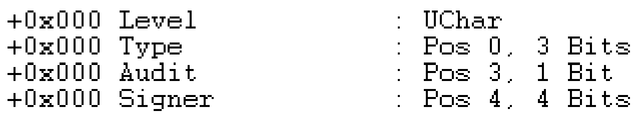
```
                public PsIsProtectedProcess
PsIsProtectedProcess proc near          ; DATA XREF: .pdata:00000001403A2A60↓o
                test    byte ptr [rcx+6CAh], 7
                mov     eax, 0
                setnbe  al
                retn
PsIsProtectedProcess endp
```

Disassembly for PsIsProtectedProcess (NTOSKRNL).
Looking at the disassembly of PsIsProtectedProcess, we can see that the TEST instruction is being used. The TEST instruction is used for a “bitwise operation”. However, we can also see that [RCX+6CAh] is the target. The PsIsProtectedProcess routine takes in one parameter only and it returns a BOOLEAN (UCHAR) – the parameter’s data-type should be a pointer to the EPROCESS structure for the target process being checked on. This tells us that the value stored in the RCX register will be the address of the PEPROCESS (EPROCESS*) for the target process, and it’s accessing the structure to read the value stored under an unknown field which symbolises if the process is or is not protected. The offset for where the field under the EPROCESS structure is located is 6CAh. This means that if you add on 0x6CA from the base address of the EPROCESS* for a process, you will land yourself at the address in which the value being checked in this routine is located at (for this environment only because the offsets regularly shift around and will vary between environment – due to patch updates and separate OS versions).

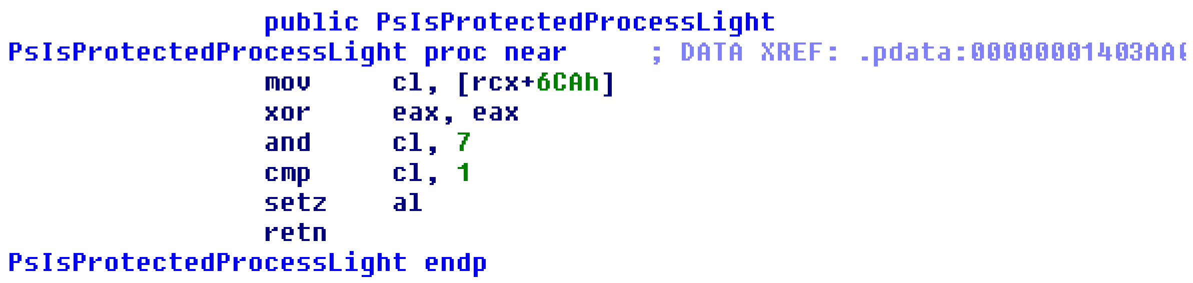
We can check with WinDbg which field is for the 0xC6A offset.


```
+0x6c8 SignatureLevel    : UChar
+0x6c9 SectionSignatureLevel : UChar
+0x6ca Protection        : _PS_PROTECTION
+0x6cb HangCount         : Pos 0, 4 Bits
+0x6cb GhostCount        : Pos 4, 4 Bits
```

WinDbg command (dt) for the _EPROCESS structure, showing the Protection field.
Nice! The field in the EPROCESS structure which holds data regarding process protection is named Protection and has a data-type of _PS_PROTECTION (which is a structure) – at-least for the standard process protection mechanism, we are yet to check on the Light variant. We can take a look at the _PS_PROTECTION structure with the dt instruction.


```
+0x000 Level            : UChar
+0x000 Type             : Pos 0, 3 Bits
+0x000 Audit            : Pos 3, 1 Bit
+0x000 Signer           : Pos 4, 4 Bits
```

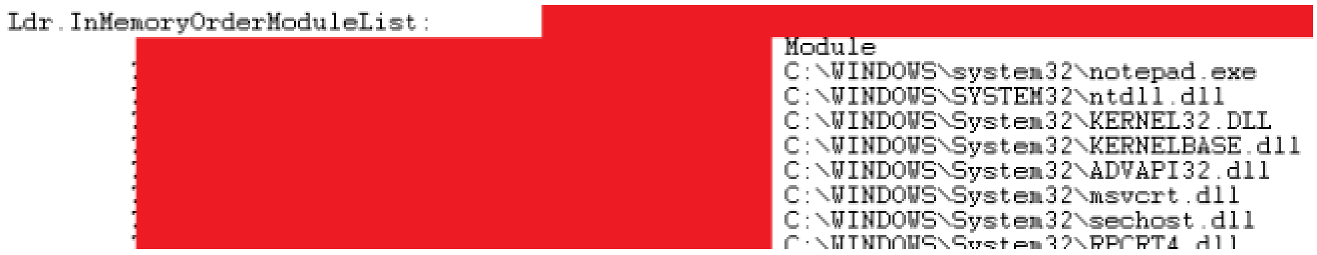
WinDbg command (dt) for the _PS_PROTECTION structure.
Now if we check the disassembly of the PsIsProtectedProcessLight routine, we can see if it uses the same mechanism to query the status.


```
                 public PsIsProtectedProcessLight
PsIsProtectedProcessLight proc near      ; DATA XREF: .pdata:00000001403AA(
                 mov     cl, [rcx+6CAh]
                 xor     eax, eax
                 and     cl, 7
                 cmp     cl, 1
                 setz    al
                 retn
PsIsProtectedProcessLight endp
```

Disassembly for PsIsProtectedProcessLight (NTOSKRNL).
It's targeting the Protection field of the EPROCESS structure as well – the same field of the structure too. The only difference here is that PsIsProtectedProcess is and PsIsProtectedProcessLight are doing some different checks.

In the PEB structure, there's an entry named Ldr which has a data-type of _PEB_LDR_DATA. Within this structure, we have a field named InMemoryOrderModuleList which has a data-type of _LIST_ENTRY. Double linked lists are very common in Windows components such as in the Windows Kernel or lower-level user-mode components.

There's an instruction in WinDbg named !peb which can be used to enumerate data for the PEB of the currently debugged process. Below is an image of what the output will look like, focus only on the non-highlighted parts.



```
Ldr.InMemoryOrderModuleList: [redacted]
            [redacted]          Module
                                C:\WINDOWS\system32\notepad.exe
                                C:\WINDOWS\SYSTEM32\ntdll.dll
                                C:\WINDOWS\System32\KERNEL32.DLL
                                C:\WINDOWS\System32\KERNELBASE.dll
                                C:\WINDOWS\System32\ADVAPI32.dll
                                C:\WINDOWS\System32\msvcrt.dll
                                C:\WINDOWS\System32\sechost.dll
                                C:\WINDOWS\System32\RPCRT4.dll
```

WinDbg command (!peb) output.
If we go through the InMemoryOrderModuleList, we can extract each entry and assign to a pointer of the LDR_DATA_TABLE_ENTRY structure using the CONTAINING_RECORD macro. Then we could view details about the current module enumerated using the linked lists… We will do this during the practical code section which is right about now.

We're going to be using the PEB for practical use in the next section.

User-Mode

In this section we're going to be re-writing a few Win32 API routines in user-mode which rely on the Process Environment Block.

1. GetModuleHandle – using the Ldr field of the PEB structure
2. GetModuleFileName – using the ProcessParameters field of the PEB structure

We need to make sure we've declared some structures. Depending on the header files you're using, you may not need them. However if you do need them…

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    WCHAR *Buffer;
} UNICODE_STRING, PUNICODE_STRING;

typedef const UNICODE_STRING
              *PCUNICODE_STRING;

typedef struct _CLIENT_ID {
    PVOID UniqueProcess;
    PVOID UniqueThread;
} CLIENT_ID, *PCLIENT_ID;

typedef struct _RTL_USER_PROCESS_PARAMETERS {
    BYTE Reserved1[16];
    PVOID Reserved2[10];
    UNICODE_STRING ImagePathName;
    UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;

typedef struct _PEB_LDR_DATA {
    BYTE Reserved1[8];
    PVOID Reserved2[3];
    LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID BaseAddress;
    PVOID Reserved3[2];
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
#pragma warning(push)
#pragma warning(disable: 4201) // we'll always use the Microsoft compiler
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    } DUMMYUNIONNAME;
#pragma warning(pop)
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;

typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID Reserved4[3];
    PVOID AtlThunkSListPtr;
    PVOID Reserved5;
    ULONG Reserved6;
    PVOID Reserved7;
    ULONG Reserved8;
} PEB, *PPEB;

typedef struct _TEB {
    NT_TIB NtTib;
    PVOID EnvironmentPointer;
    CLIENT_ID ClientId;
    PVOID ActiveRpcHandle;
    PVOID ThreadLocalStoragePointer;
    PPEB ProcessEnvironmentBlock;
} TEB, *PTEB;
```

The next thing you might want is a global definition for NtCurrentPeb(). This isn't mandatory but it can be a bit helpful if you'd prefer to type NtCurrentPeb() instead of NtCurrentTeb()->ProcessEnvironmentBlock every-time you need to gain access to the PEB. I always preferred to type NtCurrentPeb() but that's just me.

```
#define NtCurrentPeb() \
        NtCurrentTeb()->ProcessEnvironmentBlock
```

What is NtCurrentTeb()?

NtCurrentTeb() is a function which is packed within winnt.h, and it'll return a pointer to the TEB structure at the correct address of where the TEB is located.

NtCurrentTeb() will change depending on the configuration however for a 32-bit compilation, it will locate the TEB by using the __readfsdword macro, targeting 0x18 as the location. This means that the target location is actually FS:[0x18]. For a 64-bit compilation, __readgsqword will be used and the target location will be different.

GetModuleHandle replacement

```
HMODULE GetModuleHandleWrapper(
    WCHAR *ModuleName
)
{
    PPEB ProcessEnvironmentBlock = NtCurrentPeb();
    PPEB_LDR_DATA PebLdrData = { 0 };
    PLDR_DATA_TABLE_ENTRY LdrDataTableEntry = { 0 };
    PLIST_ENTRY ModuleList = { 0 },
                ForwardLink = { 0 };

    if (ProcessEnvironmentBlock)
    {
        PebLdrData = ProcessEnvironmentBlock->Ldr;

        if (PebLdrData)
        {
            ModuleList = &PebLdrData->InMemoryOrderModuleList;
            ForwardLink = ModuleList->Flink;

            while (ModuleList != ForwardLink)
            {
                LdrDataTableEntry = CONTAINING_RECORD(ForwardLink,
                    LDR_DATA_TABLE_ENTRY,
                    InMemoryOrderLinks);

                if (LdrDataTableEntry)
                {
                    if (LdrDataTableEntry->BaseDllName.Buffer)
                    {
                        if (!_wcsicmp(LdrDataTableEntry->BaseDllName.Buffer,
                            ModuleName))
                        {
                            return (HMODULE)LdrDataTableEntry->BaseAddress;
                        }
                     }
                 }

                 ForwardLink = ForwardLink->Flink;
             }
         }
    }

    return 0;
}
```

The above routine does the following.

1. Retrieves the PPEB
2. Checks if the PPEB could be acquired or not
3. Enumerates the InMemoryOrderModuleList
4. Retrieves a pointer to the LDR_DATA_TABLE_ENTRY structure for each entry
5. Returns the BaseAddress of the module if its a match based on module name buffer comparison with the parameter passed in

GetModuleFileName wrapper

```
WCHAR *GetModuleFileNameWrapper()
{
    PPEB ProcessEnvironmentBlock = NtCurrentPeb();

    if (ProcessEnvironmentBlock)if (ProcessEnvironmentBlock)
    {
        if (ProcessEnvironmentBlock->ProcessParameters)
        {
            if (ProcessEnvironmentBlock->ProcessParameters->ImagePathName.Buffer)
            {
                if (ProcessEnvironmentBlock->ProcessParameters->ImagePathName.Buffer)
                {
                    return ProcessEnvironmentBlock->ProcessParameters-
>ImagePathName.Buffer;
                }
            }
        }
    }

    return NULL;
}
```

The above routine does the following.

1. Retrieves the PPEB (pointer to the PEB)
2. Checks if the PPEB could be acquired or not
3. Checks if it can access the ProcessParameters field
4. Returns the ImagePathName buffer (it’s a UNICODE_STRING so the Buffer field is a wchar_t*)

All of this has been known for an extremely long time now but for those of you which have only just got into Windows Internals and started studying areas like the Process Environment Block, this could help clear things up for you quickly and put an end to some confusion.

As always, thanks for reading.

NtOpcode