Raytheon
Blackbird Technologies

Proof-of-Concept (PoC) Report
Anti-Debugging and Anti-Emulation

For

SIRIUS Task Order PIQUE

Submitted to:

U.S. Government

Submitted by:

Raytheon Blackbird Technologies, Inc.
13900 Lincoln Park Drive
Suite 400
Herndon, VA 20171

07 August 2015

This document includes data that shall not be disclosed outside the Government and shall not be duplicated, used, or disclosed—in whole or in part—for any purpose other than to evaluate this concept. If, however, a contract is awarded to Blackbird as a result of—or in connection with—the submission of these data, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the resulting contract. This restriction does not limit the Government's right to use information contained in these data if they are obtained from another source without restriction.
This document contains commercial or financial information, or trade secrets, of Raytheon Blackbird Technologies, Inc. that are confidential and exempt from disclosure to the public under the Freedom of Information Act, 5 U.S.C. 552(b)(4), and unlawful disclosure thereof is a violation of the Trade Secrets Act, 18 U.S.C. 1905. Public disclosure of any such information or trade secrets shall not be made without the prior written permission of Raytheon Blackbird Technologies, Inc.

(U) Table of Contents

(U) Executive Summary

(U) This report is an overview of techniques for detecting debuggers and emulation enviornments. This report represents a PoC delivery for July.

(U) Anti-Debugging

(U) There are a number of techniques to determine if a debugger is attached, including the use of Windows APIs, manually checking memory for debugger artifacts, and searching the system for forensics evidence of a debugger.

(U) Using Windows APIs

(U) The following Windows APIs can be used to determine if the application is being debugged:

IsDebuggerPresent()

CheckRemoteDebuggerPresent()

NtQueryInformationProcess()

CheckRemoteDebuggerPresent()

OutputDebugString()

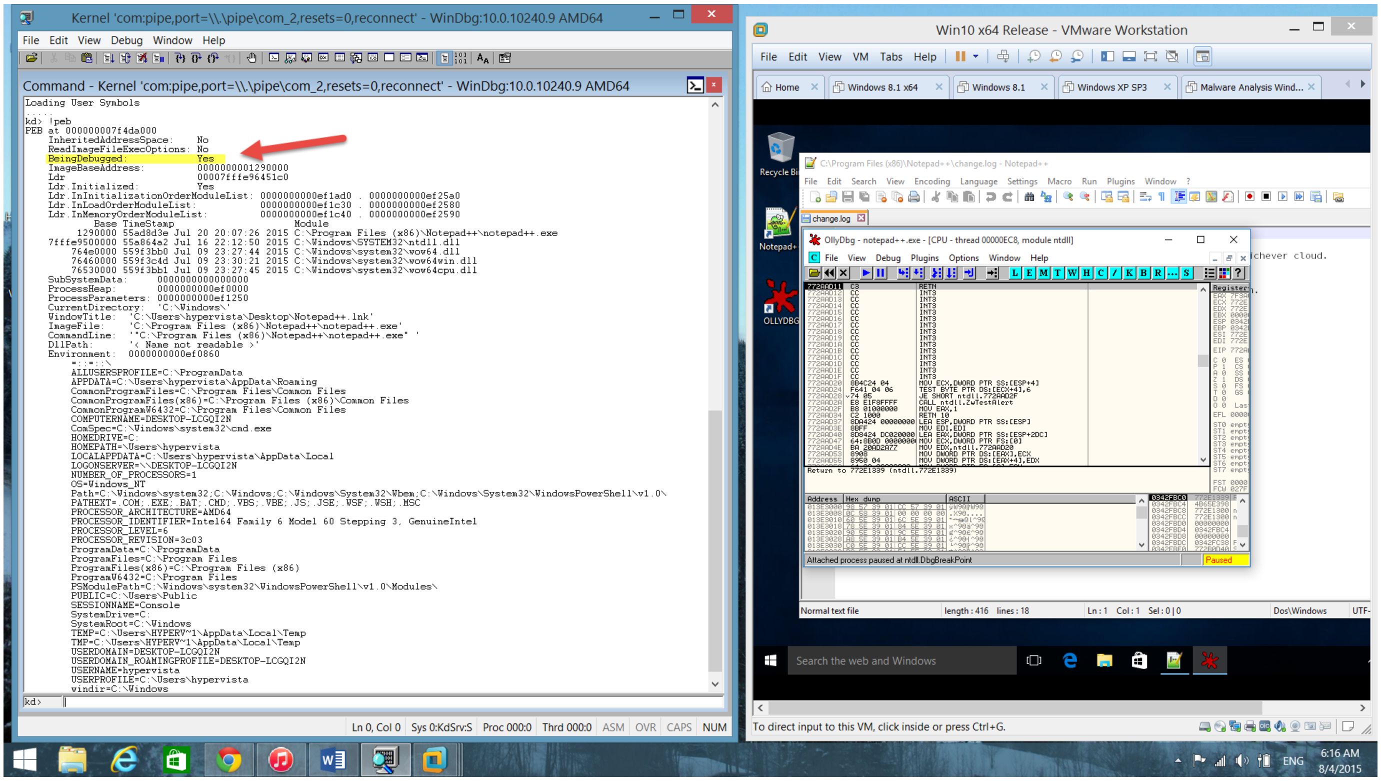
(U) These function calls are easily implemented in code. However, like many APIs, they are easily hooked to provide a false answer as to whether the application is being debugged and therefore many malware authors prefer to manually check memory structures for the presence of a debugger, which is why we don't spend time on them in this report.

(U) Manually Checking Memory Structures

(U) Because Windows APIs can be hooked to return false information about whether or not the application is being debugged, it is sometimes preferable to manually check structures such as the PEB to determine if a debugger is present.

(U) Checking the PEB BeingDebugged Flag

(U) Windows maintains a Process Execution Block (PEB) structure for each process running. The PEB contains all user-mode parameters of the running process, including a flag relating to whether the process is being debugged or not (BeingDebugged). Figure 1 shows the Windows 10 x64 Enterprise PEB structure with the Notepad++ application being debugged using Ollydbg. The BeingDebugged flag can be checked programmatically by malware.

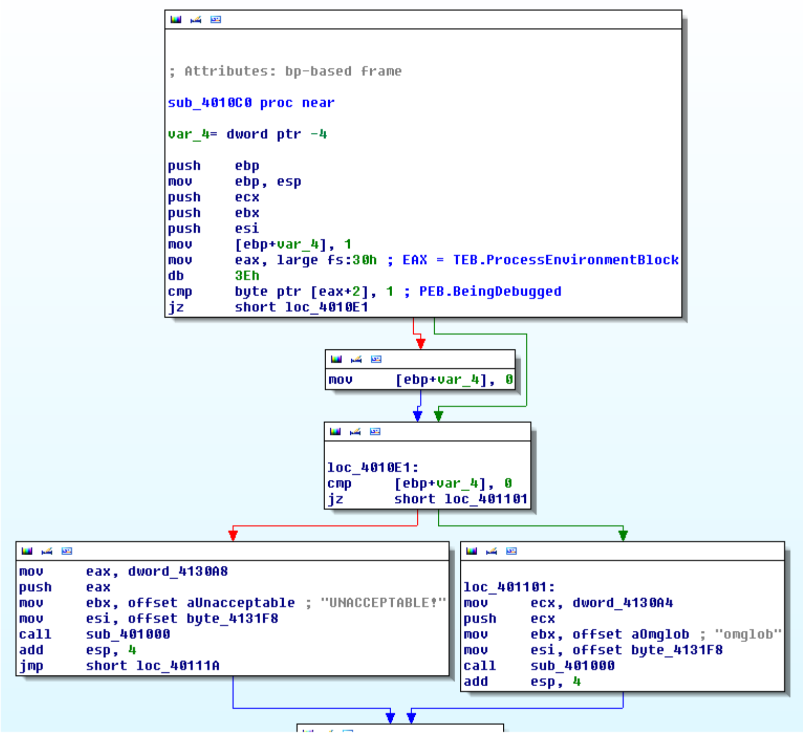

Kernel 'com:pipe,port=\\.\pipe\com_2,resets=0,reconnect' - WinDbg:10.0.10240.9 AMD64
File Edit View Debug Window Help
Command - Kernel 'com:pipe,port=\\.\pipe\com_2,resets=0,reconnect' - WinDbg:10.0.10240.9 AMD64
Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM
Win10 x64 Release - VMware Workstation
File Edit View VM Tabs Help
Home
Windows 8.1 x64
Windows 8.1
Windows XP SP3
Malware Analysis Wind...
Recycle Bin
Notepad++
OLLYDBG
C:\Program Files (x86)\Notepad++\change.log - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
change.log
OllyDbg - notepad++.exe - [CPU - thread 00000EC8, module ntdll]
File View Debug Plugins Options Window Help
Attached process paused at ntdll.DbgBreakPoint
Paused
Normal text file
length : 416 lines : 18
Ln : 1 Col : 1 Sel : 0 | 0
Dos\Windows
Search the web and Windows
To direct input to this VM, click inside or press Ctrl+G.
6:16 AM
8/4/2015

```
Loading User Symbols
.....
kd> !peb
PEB at 000000007f4da000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         0000000001290000
    Ldr                       00007fffe96451c0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 0000000000ef1ad0 . 0000000000ef25a0
    Ldr.InLoadOrderModuleList:           0000000000ef1c30 . 0000000000ef2580
    Ldr.InMemoryOrderModuleList:         0000000000ef1c40 . 0000000000ef2590
            Base TimeStamp                     Module
         1290000 55ad8d3e Jul 20 20:07:26 2015 C:\Program Files (x86)\Notepad++\notepad++.exe
    7fffe9500000 55a864a2 Jul 16 22:12:50 2015 C:\Windows\SYSTEM32\ntdll.dll
        764e0000 559f3bb0 Jul 09 23:27:44 2015 C:\Windows\system32\wow64.dll
        76460000 559f3c4d Jul 09 23:30:21 2015 C:\Windows\system32\wow64win.dll
        76530000 559f3bb1 Jul 09 23:27:45 2015 C:\Windows\system32\wow64cpu.dll
    SubSystemData:     0000000000000000
    ProcessHeap:       0000000000ef0000
    ProcessParameters: 0000000000ef1250
    CurrentDirectory:  'C:\Windows\'
    WindowTitle:  'C:\Users\hypervista\Desktop\Notepad++.lnk'
    ImageFile:    'C:\Program Files (x86)\Notepad++\notepad++.exe'
    CommandLine:  '"C:\Program Files (x86)\Notepad++\notepad++.exe" '
    DllPath:      '< Name not readable >'
    Environment:  0000000000ef0860
        =::=::\
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\hypervista\AppData\Roaming
        CommonProgramFiles=C:\Program Files\Common Files
        CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
        CommonProgramW6432=C:\Program Files\Common Files
        COMPUTERNAME=DESKTOP-LCGQI2N
        ComSpec=C:\Windows\system32\cmd.exe
        HOMEDRIVE=C:
        HOMEPATH=\Users\hypervista
        LOCALAPPDATA=C:\Users\hypervista\AppData\Local
        LOGONSERVER=\\DESKTOP-LCGQI2N
        NUMBER_OF_PROCESSORS=1
        OS=Windows_NT
        Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\
        PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
        PROCESSOR_ARCHITECTURE=AMD64
        PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 60 Stepping 3, GenuineIntel
        PROCESSOR_LEVEL=6
        PROCESSOR_REVISION=3c03
        ProgramData=C:\ProgramData
        ProgramFiles=C:\Program Files
        ProgramFiles(x86)=C:\Program Files (x86)
        ProgramW6432=C:\Program Files
        PSModulePath=C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
        PUBLIC=C:\Users\Public
        SESSIONNAME=Console
        SystemDrive=C:
        SystemRoot=C:\Windows
        TEMP=C:\Users\HYPERV~1\AppData\Local\Temp
        TMP=C:\Users\HYPERV~1\AppData\Local\Temp
        USERDOMAIN=DESKTOP-LCGQI2N
        USERDOMAIN_ROAMINGPROFILE=DESKTOP-LCGQI2N
        USERNAME=hypervista
        USERPROFILE=C:\Users\hypervista
        windir=C:\Windows
kd>
```

(U) Figure 1. Windows 10 x64 Enterprise PEB Structure – Application Debug

Use or disclosure of data contained on this sheet is subject to the restrictions on the title page of this document.

(U) Program flow of a routine for checking the PEB BeingDebugged flag is represented in Figure 2:



```
; Attributes: bp-based frame

sub_4010C0 proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
push    ebx
push    esi
mov     [ebp+var_4], 1
mov     eax, large fs:30h ; EAX = TEB.ProcessEnvironmentBlock
db      3Eh
cmp     byte ptr [eax+2], 1 ; PEB.BeingDebugged
jz      short loc_4010E1
```

```
mov     [ebp+var_4], 0
```

```
loc_4010E1:
cmp     [ebp+var_4], 0
jz      short loc_401101
```

```
mov     eax, dword_4130A8
push    eax
mov     ebx, offset aUnacceptable ; "UNACCEPTABLE!"
mov     esi, offset byte_4131F8
call    sub_401000
add     esp, 4
jmp     short loc_40111A
```

```
loc_401101:
mov     ecx, dword_4130A4
push    ecx
mov     ebx, offset aOmglob ; "omglob"
mov     esi, offset byte_4131F8
call    sub_401000
add     esp, 4
```

Figure 2. Programmatically Checking the BeingDebugged Flag

(U) Brief code explanation:

```
mov     eax, large fs:30h         ; move PEB structure (fs:30h) into EAX
cmp     byte ptr [eax+2], 1       ; check to see if the BeingDebugged flag is 1
jz      short loc_4010E1          ; if byte ptr [eax+2] =1 it's being debugged and jmp not made
```

(U) NOTE: there are several Ollydbg plug-in modules that change the BeingDebugged flag in order to 'trick' malware into believing it is not being debugged. The Ollydbg plug-ins that alter the BeingDebugged flag are: ***Hide Debugger, Hidedebug, and PhantOm.***

(U) Checking the PEB ProcessHeap Flag

(U) Within the PEB is a data structure called ProcessHeap located at offset 0x18 in the PEB as shown in Figure 3. The first heap (at ef0000 in this example) contains a header with fields used to tell the kernel whether the heap was created within a debugger. The flags we are interested in are ***ForceFlags*** and ***Flags***, which can be seen in Figure 4. The ForceFlags flag is a more reliable flag to check because Flags is usually either set to the value of ForceFlag or XORed with 2, as it is in our example as can be seen in Figure 4.

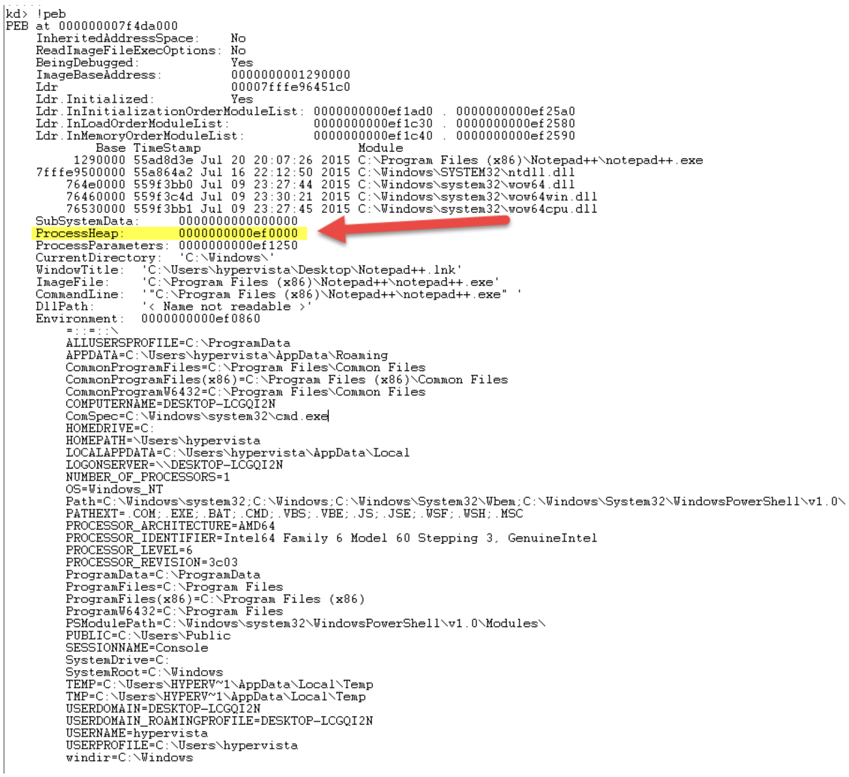
(U) The ForceFlags flag offset in the heap will differ depending on the OS version as shown in Table 1.

Table 1. ForceFlags Offset by Windows OS Version

OS Version	ForceFlags Offset
Windows XP 32-bit	0x10
Windows 7 32-bit	0x44
Windows 7 64-bit	0x74
Windows 8 32-bit	0x44
Windows 8 64-bit	0x74
Windows 10 32-bit	0x44
Windows 10 64-bit	0x74

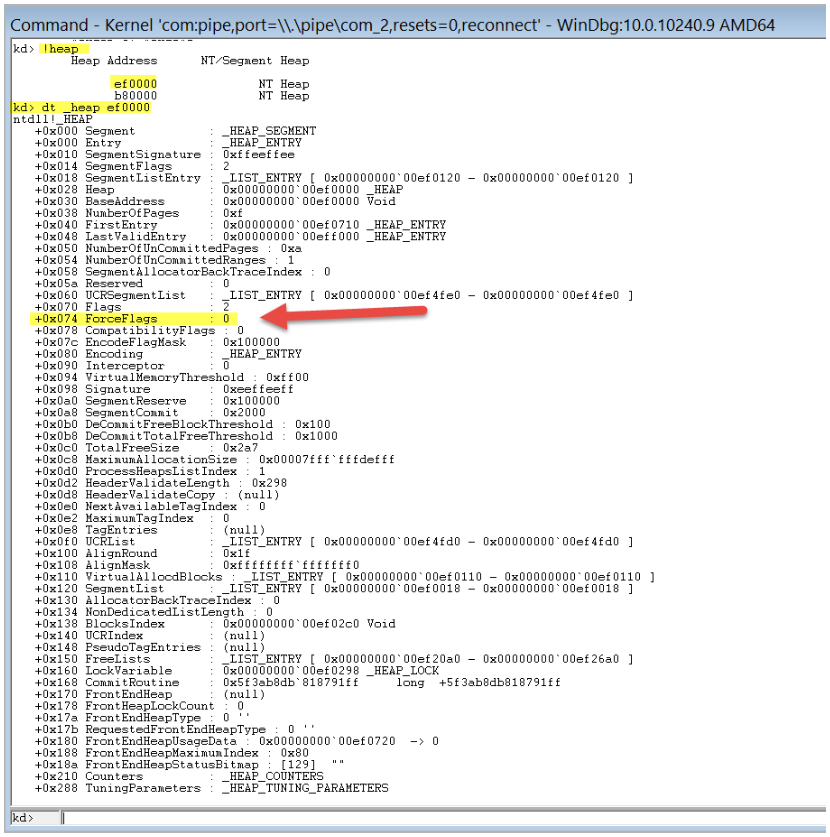
(U) An example of an assembly language routine to manually check the ForceFlags flag is:

```
mov eax, large fs:30h                    ; load PEB structure into EAX
mov eax, dword ptr [eax+18h]             ; Go to ProcessHeap offset in PEB
cmp dword ptr ds:[eax+74h], 0            ; Check to see if ForceFlags is 0 (offset varies per Table 1)
jne DebuggerDetected                     ; DebuggerDetected routine defined elsewhere
```



```
kd> !peb
PEB at 000000007f4da000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         0000000001290000
    Ldr                       00007fffe96451c0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 0000000000ef1ad0 . 0000000000ef25a0
    Ldr.InLoadOrderModuleList:           0000000000ef1c30 . 0000000000ef2580
    Ldr.InMemoryOrderModuleList:         0000000000ef1c40 . 0000000000ef2590
            Base TimeStamp                     Module
         1290000 55ad8d3e Jul 20 20:07:26 2015 C:\Program Files (x86)\Notepad++\notepad++.exe
    7fffe9500000 55a864a2 Jul 16 22:12:50 2015 C:\Windows\SYSTEM32\ntdll.dll
        764e0000 559f3bb0 Jul 09 23:27:44 2015 C:\Windows\system32\wow64.dll
        76460000 559f3c4d Jul 09 23:30:21 2015 C:\Windows\system32\wow64win.dll
        76530000 559f3bb1 Jul 09 23:27:45 2015 C:\Windows\system32\wow64cpu.dll
    SubSystemData:     0000000000000000
    ProcessHeap:       0000000000ef0000
    ProcessParameters: 0000000000ef1250
    CurrentDirectory:  'C:\Windows\'
    WindowTitle:  'C:\Users\hypervista\Desktop\Notepad++.lnk'
    ImageFile:    'C:\Program Files (x86)\Notepad++\notepad++.exe'
    CommandLine:  '"C:\Program Files (x86)\Notepad++\notepad++.exe" '
    DllPath:      '< Name not readable >'
    Environment:  0000000000ef0860
        =::=::\
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\hypervista\AppData\Roaming
        CommonProgramFiles=C:\Program Files\Common Files
        CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
        CommonProgramW6432=C:\Program Files\Common Files
        COMPUTERNAME=DESKTOP-LCGQI2N
        ComSpec=C:\Windows\system32\cmd.exe
        HOMEDRIVE=C:
        HOMEPATH=\Users\hypervista
        LOCALAPPDATA=C:\Users\hypervista\AppData\Local
        LOGONSERVER=\\DESKTOP-LCGQI2N
        NUMBER_OF_PROCESSORS=1
        OS=Windows_NT
        Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\
        PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
        PROCESSOR_ARCHITECTURE=AMD64
        PROCESSOR_IDENTIFIER=Intel64 Family 6 Model 60 Stepping 3, GenuineIntel
        PROCESSOR_LEVEL=6
        PROCESSOR_REVISION=3c03
        ProgramData=C:\ProgramData
        ProgramFiles=C:\Program Files
        ProgramFiles(x86)=C:\Program Files (x86)
        ProgramW6432=C:\Program Files
        PSModulePath=C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
        PUBLIC=C:\Users\Public
        SESSIONNAME=Console
        SystemDrive=C:
        SystemRoot=C:\Windows
        TEMP=C:\Users\HYPERV~1\AppData\Local\Temp
        TMP=C:\Users\HYPERV~1\AppData\Local\Temp
        USERDOMAIN=DESKTOP-LCGQI2N
        USERDOMAIN_ROAMINGPROFILE=DESKTOP-LCGQI2N
        USERNAME=hypervista
        USERPROFILE=C:\Users\hypervista
        windir=C:\Windows
```

Figure 3. ProcessHeap in the PEB at offset 0x18



Command - Kernel 'com:pipe,port=\\.\pipe\com_2,resets=0,reconnect' - WinDbg:10.0.10240.9 AMD64

```
kd> !heap
        Heap Address      NT/Segment Heap

               ef0000              NT Heap
               b80000              NT Heap
kd> dt _heap ef0000
ntdll!_HEAP
   +0x000 Segment          : _HEAP_SEGMENT
   +0x000 Entry            : _HEAP_ENTRY
   +0x010 SegmentSignature : 0xffeeffee
   +0x014 SegmentFlags     : 2
   +0x018 SegmentListEntry : _LIST_ENTRY [ 0x00000000`00ef0120 - 0x00000000`00ef0120 ]
   +0x028 Heap             : 0x00000000`00ef0000 _HEAP
   +0x030 BaseAddress      : 0x00000000`00ef0000 Void
   +0x038 NumberOfPages    : 0xf
   +0x040 FirstEntry       : 0x00000000`00ef0710 _HEAP_ENTRY
   +0x048 LastValidEntry   : 0x00000000`00eff000 _HEAP_ENTRY
   +0x050 NumberOfUnCommittedPages : 0xa
   +0x054 NumberOfUnCommittedRanges : 1
   +0x058 SegmentAllocatorBackTraceIndex : 0
   +0x05a Reserved         : 0
   +0x060 UCRSegmentList   : _LIST_ENTRY [ 0x00000000`00ef4fe0 - 0x00000000`00ef4fe0 ]
   +0x070 Flags            : 2
   +0x074 ForceFlags       : 0
   +0x078 CompatibilityFlags : 0
   +0x07c EncodeFlagMask   : 0x100000
   +0x080 Encoding         : _HEAP_ENTRY
   +0x090 Interceptor      : 0
   +0x094 VirtualMemoryThreshold : 0xff00
   +0x098 Signature        : 0xeeffeeff
   +0x0a0 SegmentReserve   : 0x100000
   +0x0a8 SegmentCommit    : 0x2000
   +0x0b0 DeCommitFreeBlockThreshold : 0x100
   +0x0b8 DeCommitTotalFreeThreshold : 0x1000
   +0x0c0 TotalFreeSize    : 0x2a7
   +0x0c8 MaximumAllocationSize : 0x00007fff`fffdefff
   +0x0d0 ProcessHeapsListIndex : 1
   +0x0d2 HeaderValidateLength : 0x298
   +0x0d8 HeaderValidateCopy : (null)
   +0x0e0 NextAvailableTagIndex : 0
   +0x0e2 MaximumTagIndex  : 0
   +0x0e8 TagEntries       : (null)
   +0x0f0 UCRList          : _LIST_ENTRY [ 0x00000000`00ef4fd0 - 0x00000000`00ef4fd0 ]
   +0x100 AlignRound       : 0x1f
   +0x108 AlignMask        : 0xffffffff`fffffff0
   +0x110 VirtualAllocdBlocks : _LIST_ENTRY [ 0x00000000`00ef0110 - 0x00000000`00ef0110 ]
   +0x120 SegmentList      : _LIST_ENTRY [ 0x00000000`00ef0018 - 0x00000000`00ef0018 ]
   +0x130 AllocatorBackTraceIndex : 0
   +0x134 NonDedicatedListLength : 0
   +0x138 BlocksIndex      : 0x00000000`00ef02c0 Void
   +0x140 UCRIndex         : (null)
   +0x148 PseudoTagEntries : (null)
   +0x150 FreeLists        : _LIST_ENTRY [ 0x00000000`00ef20a0 - 0x00000000`00ef26a0 ]
   +0x160 LockVariable     : 0x00000000`00ef0298 _HEAP_LOCK
   +0x168 CommitRoutine    : 0x5f3ab8db`818791ff     long  +5f3ab8db818791ff
   +0x170 FrontEndHeap     : (null)
   +0x178 FrontHeapLockCount : 0
   +0x17a FrontEndHeapType : 0 ''
   +0x17b RequestedFrontEndHeapType : 0 ''
   +0x180 FrontEndHeapUsageData : 0x00000000`00ef0720  -> 0
   +0x188 FrontEndHeapMaximumIndex : 0x80
   +0x18a FrontEndHeapStatusBitmap : [129]  ""
   +0x210 Counters         : _HEAP_COUNTERS
   +0x288 TuningParameters : _HEAP_TUNING_PARAMETERS

kd>
```

Figure 4. ForceFlags Flag at offset 0x74 (Windows 10 64-bit)

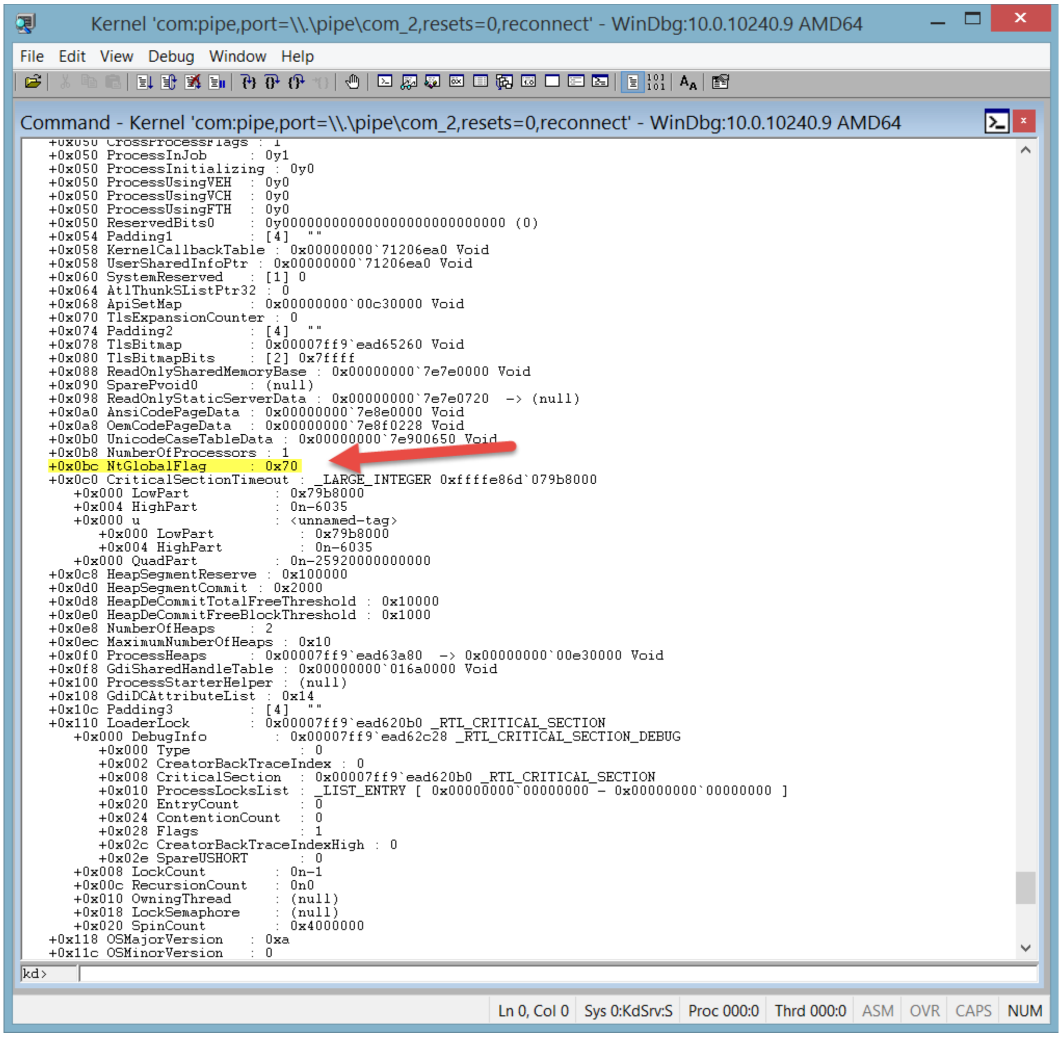
(U) Checking the PEB NTGlobalFlag Value

(U) Another PEB element that contains information about whether the application was launched under a debugger is the NTGlobalFlag. The NTGlobalFlag element offset in the PEB differs between the 32-bit and 64-bit versions of Windows, as shown in Table 2.

Table 2. NTGlobalFlag Offset in PEB by OS Variety

Windows Variety	NTGlobalFlag Offset in PEB
32-bit Versions (Win 8, 8.1, and 10)	0x068
64-bit Versions (Win 8, 8.1, and 10)	0x0bc

(U) If the value of the NTGlobalFlag is 0x70, the application is being debugged as can be seen in Figure 5. The value of 0x70 is a combination of three separate flags being set when a heap is created by a debugger. The three flags set when a heap is created by a debugger are FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20), and FLG_HEAP_VALIDATE_PARAMETERS (0x40).


Kernel 'com:pipe,port=\\.\pipe\com_2,resets=0,reconnect' - WinDbg:10.0.10240.9 AMD64
File
Edit
View
Debug
Window
Help
Command - Kernel 'com:pipe,port=\\.\pipe\com_2,resets=0,reconnect' - WinDbg:10.0.10240.9 AMD64
kd>
Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM

```
   +0x050 CrossProcessFlags : 1
   +0x050 ProcessInJob     : 0y1
   +0x050 ProcessInitializing : 0y0
   +0x050 ProcessUsingVEH  : 0y0
   +0x050 ProcessUsingVCH  : 0y0
   +0x050 ProcessUsingFTH  : 0y0
   +0x050 ReservedBits0    : 0y000000000000000000000000000 (0)
   +0x054 Padding1         : [4]  ""
   +0x058 KernelCallbackTable : 0x00000000`71206ea0 Void
   +0x058 UserSharedInfoPtr : 0x00000000`71206ea0 Void
   +0x060 SystemReserved   : [1] 0
   +0x064 AtlThunkSListPtr32 : 0
   +0x068 ApiSetMap        : 0x00000000`00c30000 Void
   +0x070 TlsExpansionCounter : 0
   +0x074 Padding2         : [4]  ""
   +0x078 TlsBitmap        : 0x00007ff9`ead65260 Void
   +0x080 TlsBitmapBits    : [2] 0x7ffff
   +0x088 ReadOnlySharedMemoryBase : 0x00000000`7e7e0000 Void
   +0x090 SparePvoid0      : (null)
   +0x098 ReadOnlyStaticServerData : 0x00000000`7e7e0720  -> (null)
   +0x0a0 AnsiCodePageData : 0x00000000`7e8e0000 Void
   +0x0a8 OemCodePageData  : 0x00000000`7e8f0228 Void
   +0x0b0 UnicodeCaseTableData : 0x00000000`7e900650 Void
   +0x0b8 NumberOfProcessors : 1
   +0x0bc NtGlobalFlag     : 0x70
   +0x0c0 CriticalSectionTimeout : _LARGE_INTEGER 0xffffe86d`079b8000
      +0x000 LowPart          : 0x79b8000
      +0x004 HighPart         : 0n-6035
      +0x000 u                : <unnamed-tag>
         +0x000 LowPart          : 0x79b8000
         +0x004 HighPart         : 0n-6035
      +0x000 QuadPart         : 0n-25920000000000
   +0x0c8 HeapSegmentReserve : 0x100000
   +0x0d0 HeapSegmentCommit : 0x2000
   +0x0d8 HeapDeCommitTotalFreeThreshold : 0x10000
   +0x0e0 HeapDeCommitFreeBlockThreshold : 0x1000
   +0x0e8 NumberOfHeaps    : 2
   +0x0ec MaximumNumberOfHeaps : 0x10
   +0x0f0 ProcessHeaps     : 0x00007ff9`ead63a80  -> 0x00000000`00e30000 Void
   +0x0f8 GdiSharedHandleTable : 0x00000000`016a0000 Void
   +0x100 ProcessStarterHelper : (null)
   +0x108 GdiDCAttributeList : 0x14
   +0x10c Padding3         : [4]  ""
   +0x110 LoaderLock       : 0x00007ff9`ead620b0 _RTL_CRITICAL_SECTION
      +0x000 DebugInfo        : 0x00007ff9`ead62c28 _RTL_CRITICAL_SECTION_DEBUG
         +0x000 Type             : 0
         +0x002 CreatorBackTraceIndex : 0
         +0x008 CriticalSection  : 0x00007ff9`ead620b0 _RTL_CRITICAL_SECTION
         +0x010 ProcessLocksList : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
         +0x020 EntryCount       : 0
         +0x024 ContentionCount  : 0
         +0x028 Flags            : 1
         +0x02c CreatorBackTraceIndexHigh : 0
         +0x02e SpareUSHORT      : 0
      +0x008 LockCount        : 0n-1
      +0x00c RecursionCount   : 0n0
      +0x010 OwningThread     : (null)
      +0x018 LockSemaphore    : (null)
      +0x020 SpinCount        : 0x4000000
   +0x118 OSMajorVersion   : 0xa
   +0x11c OSMinorVersion   : 0
```

Figure 5. NTGlobalFlag – 64-bit Application Being Debugged

Use or disclosure of data contained on this sheet is subject to the restrictions on the title page of this document.

(U) An assembly language example of manually checking the value of the NTGlobalFlag is:

```
mov eax, large fs:30h                  ; move PEB structure into EAX
cmp dword ptr ds:[eax+bch], 70h        ; check to see if NTGlobalFlag is 0x70
jz DebuggerDetected                    ; DebuggerDetected function defined elsewhere
```

(U) Identifying Debugger Behavior

(U) Because some debugging activities by necessity modify the code at debugger runtime, these alterations of the code can be detected. Debugger generated code modifications include insertion of INT instructions (not just INT 3), these INT instructions can be scanned for. Malware can use checksums on their code to determine if the running code has been altered in any way, presumably by a debugger. Lastly, malware can also perform timing checks because processes run slower when being debugged.

(U) Checking to See if SeDebugPrivilege is Set

(U) By default, a process has SeDebugPrivilege disabled. When the process is loaded by a debugger, SeDebugPrivilege is enabled. Malware will check to see if SeDebugPrivilege has been enabled by trying to open the CSRSS.EXE process and if it is able to open it the process is running under a debugger.

(U) Scanning for INT

(U) INT 3 is a software interrupt that debuggers insert into running code, replacing an existing instruction, which calls the debug exception handler, i.e. sets a breakpoint. The opcode for INT 3 is 0xcc. In addition to inserting an INT 3 opcode into running code, debuggers also insert INT <immediate value> in some cases. The opcode for INT <immediate value> is 0xcd. A common method malware uses to detect if it’s running under the control of a debugger is to scan for these opcodes. A rough assembly language routine to scan for INT 3 is:

```
mov ecx, 400h             ; loop counter
mov eax, 0cch             ; INT 3 (0cch) search value
repne scasb               ; string search command (look for INT 3 (0cch))
jz DebuggerDetected       ; DebuggerDetected function is defined elsewhere
```

(U) Code Checksums

(U) Some malware samples calculates checksums of specific sections of its code, either CRC or MD5 to detect debugger modification of the code to implement breakpoints. This technique is less common than INT scanning, but just as effective.

(U) Timing Checks

(U) Because processes run substantially slower under a debugger (think single-stepping through code), timing checks is a very effective and popular way malware authors check for the presence of a debugger. There are a few methods for conducting timing checks for the presence of a debugger:

Take a timestamp, perform some specific operations and take another timestamp, calculate the time difference and make a judgment about whether or not the time difference is outside the bounds of normal time to conduct the operations.

Take a timestamp before and after causing an exception, calculate the time difference and make a judgment about whether the time to respond to the exception is well outside the normal time require to handle an exception.

(U) The use of the rdtsc command is quite popular among malware authors as a time checking function. In addition to rdtsc, the Windows APIs QueryPerformanceCounter() and GetTickCount are also heavily used as well.

(U) Checking the Number of Kernel DebugObjects

(U) When an application is being debugged, a DebugObject is created in the kernel. The number and type of Objects created can be retrieved via the NtQueryObject() API, which returns an OBJECT_ALL_INFORMATION structure. The OBJECT_ALL_INFORMATION structure is searched for the string, “DebugObject” and is checked for a non-zero value, indicating the presence of a debugger.

(U) Checking for a Debugger Window

(U) Most debuggers create windows, which can be detected. For example, Windbg creates WinDbgFrameClass and Ollydbg creates OLLYDBG. The Windows APIs FindWindow() and FindWindowEx() can be used to search for the debugger windows, indicating the presence of a debugger.

(U) Providing an Invalid ASCII String to OutputDebugStringA

(U) Calling OutputDebugStringA() with an invalid ASCII string will normally return a value of 1. If the process is being run under the control of a debugger, the return value when providing an invalid ASCII string to OutputDebugStringA() is the address of the string passed in as a parameter, indicating the presence of a debugger.

(U) Using the Stack Segment Register and Checking Trapflag

(U) This is an anti-tracing technique. If a debugger is tracing over a sequence of instructions that includes pop ss and pushf instructions, the debugger will not be able to unset the trapflag in the pushed value on the stack. The protection checks for the trapflag and if set it indicates the presence of a debugger. For example:

```
push    ss
pop     ss
pushf
```

(U) When tracing over the pop ss instruction in this example code, the next instruction will be executed but the debugger will not break on it, therefore stopping on the following instruction, nop in this case. This rare anti-debugging trick has been seen in the wild as follows:

```
push     ss
; junk code here
pop      ss
pushf
; junk code here
pop      eax
and      eax,100h
or       eax,eax
```

(U) The trick here is that if the debugger is tracing over this sequence of instructions, popf will be executed implicitly and the debugger will not be able to unset the trapflag. The malware checks for the trapflag and debugger is present if found.

(U) Trolling the Debugger

(U) In addition to detecting the presence of a debugger and responding accordingly, by either exiting without installing malicious code or by presenting benign behavior, some malware has been observed interfering with debugger functionality to make a malware analyst's job more difficult. While we're pretty certain the Sponsor would not use such tactics, we're including these techniques for completeness.

(U) Modifying the SEH Chain

(U) Modifying the SEH chain can be used as an anti-disassembly technique as well as an anti-debugging technique. Exception-based detection relies on the fact that debuggers will trap the exception and not immediately pass it to the process for handling. If the debugger fails to pass the exception to the process for handling, as most will do, that can be detected within the exception-handling mechanism and a determination that a debugger is running can be made.

(U) Inserting INT Commands

(U) This technique is more an obnoxious annoyance than anything. Some malware samples have been observed inserting INT 3 commands into code to cause the debugger to break, or in some cases stop all together.

(U) Inserting an INT 1 will invoke single-step mode inside the debugger.

(U) Some malware specimen insert INT 2D commands to cause the kernel debugger to break, similar to the insertion of INT 3 into application code.

(U) Following is an assembly language example of malware inserting an INT 3 command to check for the presence of a debugger. This example sets EAX to 0xffffffff inside the exception handler to signify the exception handler had been called. If EAX is not 0xffffffff after the INT 3 call, then a debugger is present.

```
;set exception handler
        push    .exception handler
        push    dword [fs:0]
        mov     [fs:0], esp

;reset flag (EAX) invoke int3
        xor     eax,eax
        int3

;restore exception handler
        pop     dword [fs:0]
        add     esp,4

;check to see if the flag has been set
        test    eax,eax
        je      .debugger_found

;exception_handler
```

(U) Inserting In-Circuit Emulator (ICE) Breakpoints

(U) Inserting an ICE breakpoint, iceb (opcode 0xf1) generates a single-step exception and the debugger will think it's a normal exception and not execute the established exception handler. Malware can take advantage of this fact by using the exception handler for its normal execution flow, which would be interrupted if a debugger is attached thereby hiding the malicious code from examination.

(U) Anti-Emulation

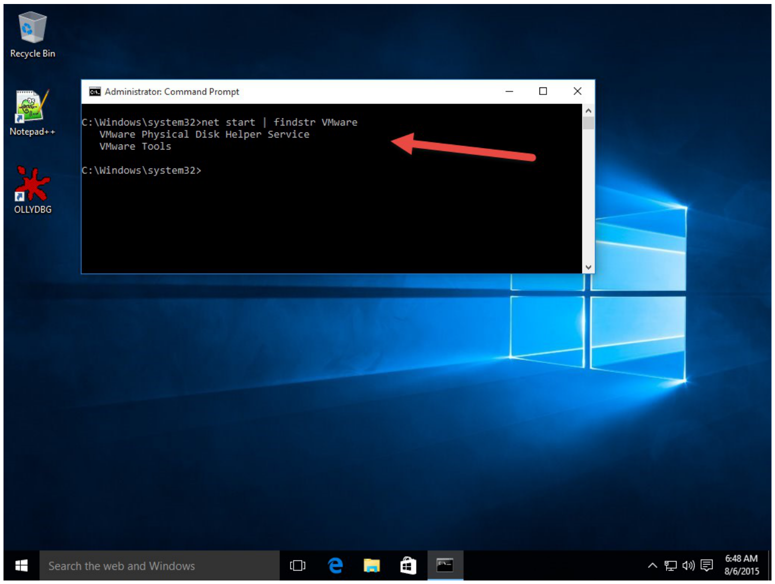
(U) Virtual environments, virtual machines and commercial sandbox technologies built upon them are collectively known as emulation environments. Emulation environments pose a risk to malware because they are used by system defenders to analyze malware at runtime. Malware authors have spent a lot of resources and written many routines to detect emulation environments in an attempt to protect their malware. However, in recent years we've seen a decrease in anti-emulation techniques in malware and attribute this trend to the fact that emulation environments have become so pervasive (think Cloud) that the presence of an emulation environment no longer automatically means it's a malware analysis platform or that it's not a valid target.

(U) Detecting VMWare Artifacts

(U) VMWare leaves many detectable artifacts when installed on a system. Malware can use these artifacts left in the file system, registry, and process listing. The following were run on a Windows 10 Virtual Machine image.

(U) Using net start | findstr VMWare

(U) A quick and easy way to determine if the VMWare Tools Service is running on the system is to open a command window and type *net start | findstr VMWare* as shown in Figure 6.



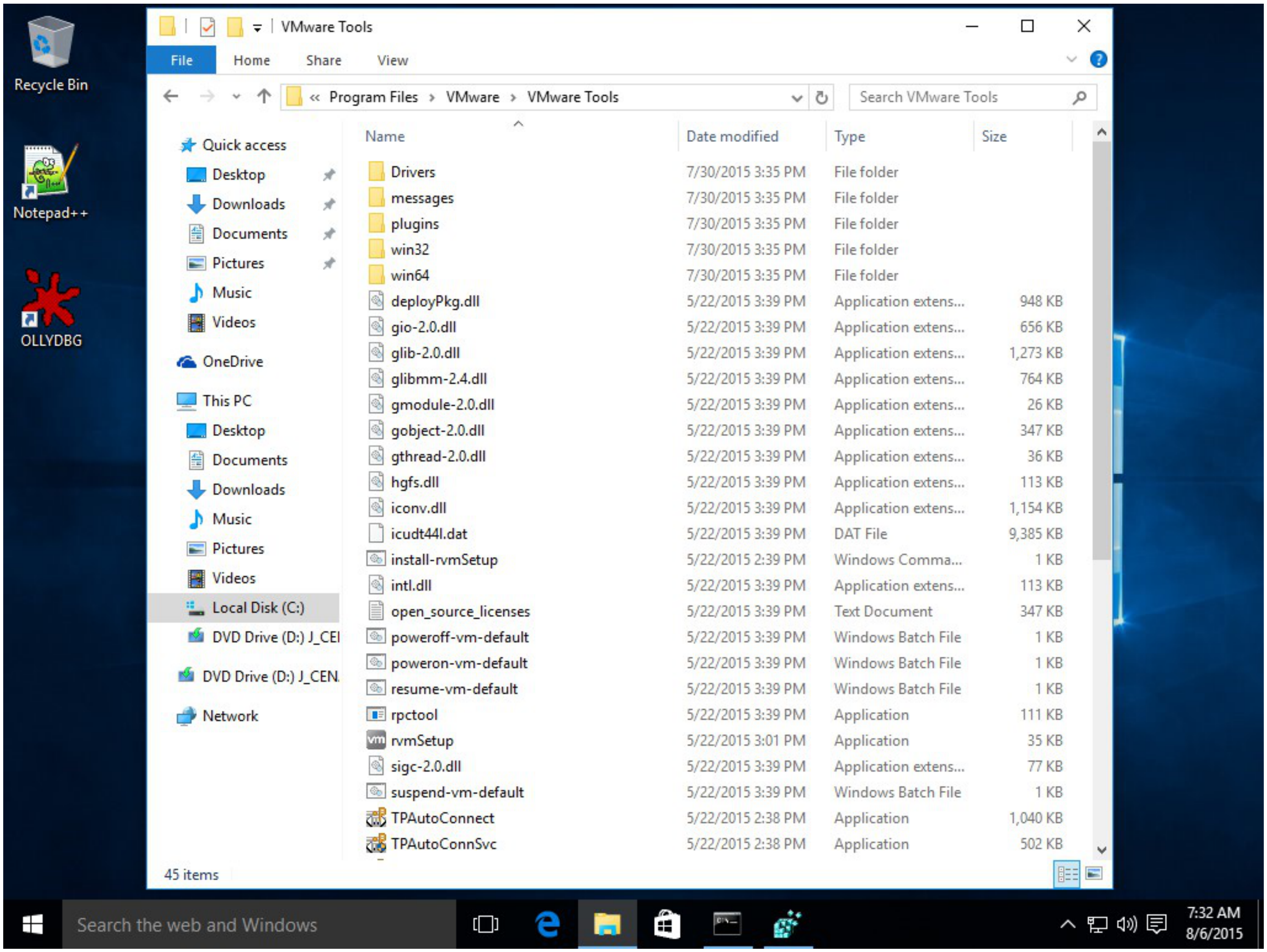
```
C:\Windows\system32>net start | findstr VMware
   VMware Physical Disk Helper Service
   VMware Tools

C:\Windows\system32>
```

Figure 6. Using cmd.exe to Find the VMWare Service

(U) Searching the File System

(U) Another quick and easy way to determine if VMWare has been installed on the target is to look in the file system (../Program Files/VMWare) as shown in Figure 7. This file system search was conducted on a Windows 10 VMWare machine, not the host machine.

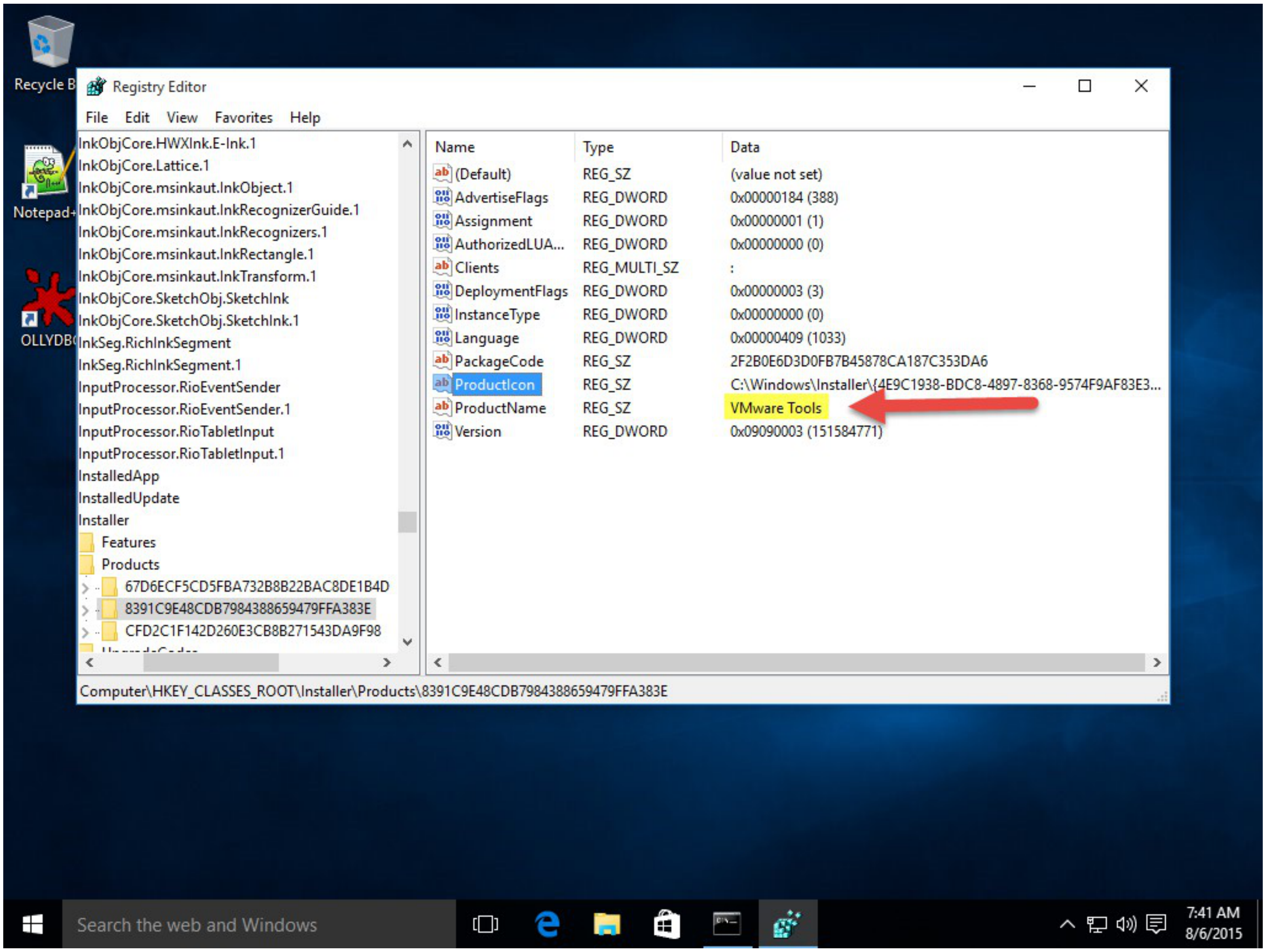


Name	Date modified	Type	Size
Drivers	7/30/2015 3:35 PM	File folder	
messages	7/30/2015 3:35 PM	File folder	
plugins	7/30/2015 3:35 PM	File folder	
win32	7/30/2015 3:35 PM	File folder	
win64	7/30/2015 3:35 PM	File folder	
deployPkg.dll	5/22/2015 3:39 PM	Application extens...	948 KB
gio-2.0.dll	5/22/2015 3:39 PM	Application extens...	656 KB
glib-2.0.dll	5/22/2015 3:39 PM	Application extens...	1,273 KB
glibmm-2.4.dll	5/22/2015 3:39 PM	Application extens...	764 KB
gmodule-2.0.dll	5/22/2015 3:39 PM	Application extens...	26 KB
gobject-2.0.dll	5/22/2015 3:39 PM	Application extens...	347 KB
gthread-2.0.dll	5/22/2015 3:39 PM	Application extens...	36 KB
hgfs.dll	5/22/2015 3:39 PM	Application extens...	113 KB
iconv.dll	5/22/2015 3:39 PM	Application extens...	1,154 KB
icudt44l.dat	5/22/2015 3:39 PM	DAT File	9,385 KB
install-rvmSetup	5/22/2015 2:39 PM	Windows Comma...	1 KB
intl.dll	5/22/2015 3:39 PM	Application extens...	113 KB
open_source_licenses	5/22/2015 3:39 PM	Text Document	347 KB
poweroff-vm-default	5/22/2015 3:39 PM	Windows Batch File	1 KB
poweron-vm-default	5/22/2015 3:39 PM	Windows Batch File	1 KB
resume-vm-default	5/22/2015 3:39 PM	Windows Batch File	1 KB
rpctool	5/22/2015 3:39 PM	Application	111 KB
rvmSetup	5/22/2015 3:01 PM	Application	35 KB
sigc-2.0.dll	5/22/2015 3:39 PM	Application extens...	77 KB
suspend-vm-default	5/22/2015 3:39 PM	Windows Batch File	1 KB
TPAutoConnect	5/22/2015 2:38 PM	Application	1,040 KB
TPAutoConnSvc	5/22/2015 2:38 PM	Application	502 KB

Figure 7. Searching the File System for VMWare

(U) Search the Registry for 'VMWare'

(U) Searching the registry for the string 'VMWare' is another simple and easy way to look for VMWare installation artifacts on the target machine. The following registry search, as shown in Figure 8, was conducted on a Windows 10 guest Virtual Machine, not the host machine.



```
InkObjCore.HWXInk.E-Ink.1
InkObjCore.Lattice.1
InkObjCore.msinkaut.InkObject.1
InkObjCore.msinkaut.InkRecognizerGuide.1
InkObjCore.msinkaut.InkRecognizers.1
InkObjCore.msinkaut.InkRectangle.1
InkObjCore.msinkaut.InkTransform.1
InkObjCore.SketchObj.SketchInk
InkObjCore.SketchObj.SketchInk.1
InkSeg.RichInkSegment
InkSeg.RichInkSegment.1
InputProcessor.RioEventSender
InputProcessor.RioEventSender.1
InputProcessor.RioTabletInput
InputProcessor.RioTabletInput.1
InstalledApp
InstalledUpdate
Installer
    Features
    Products
        67D6ECF5CD5FBA732B8B22BAC8DE1B4D
        8391C9E48CDB7984388659479FFA383E
        CFD2C1F142D260E3CB8B271543DA9F98
```

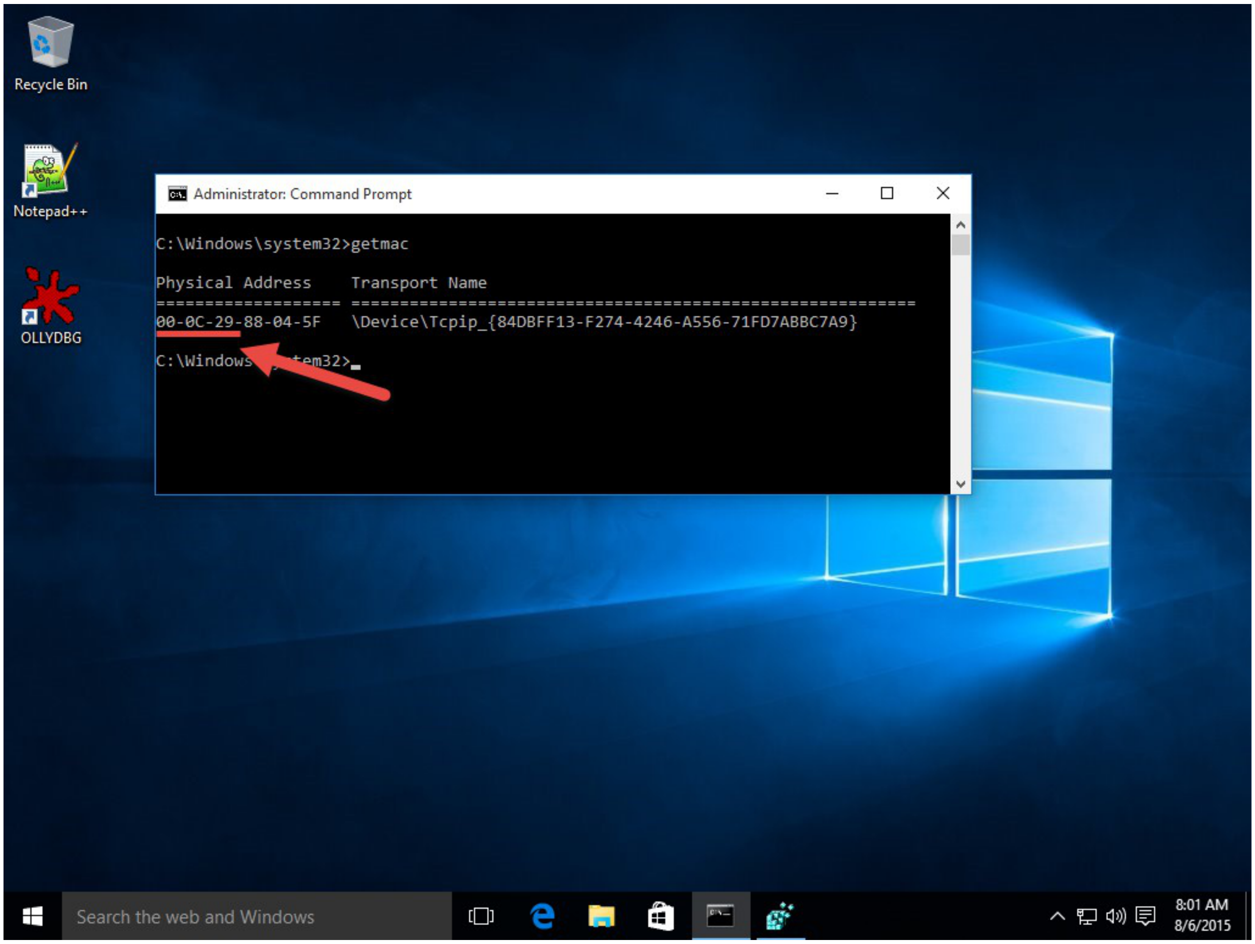
Name	Type	Data
(Default)	REG_SZ	(value not set)
AdvertiseFlags	REG_DWORD	0x00000184 (388)
Assignment	REG_DWORD	0x00000001 (1)
AuthorizedLUA...	REG_DWORD	0x00000000 (0)
Clients	REG_MULTI_SZ	:
DeploymentFlags	REG_DWORD	0x00000003 (3)
InstanceType	REG_DWORD	0x00000000 (0)
Language	REG_DWORD	0x00000409 (1033)
PackageCode	REG_SZ	2F2B0E6D3D0FB7B45878CA187C353DA6
ProductIcon	REG_SZ	C:\Windows\Installer\{4E9C1938-BDC8-4897-8368-9574F9AF83E3...
ProductName	REG_SZ	VMware Tools
Version	REG_DWORD	0x09090003 (151584771)

Computer\HKEY_CLASSES_ROOT\Installer\Products\8391C9E48CDB7984388659479FFA383E

Figure 8. Searching the Registry for 'VMWare'

(U) Checking the MAC for Leading 00:0C:29

(U) The first three groups in a MAC address identify the manufacturer of the network device. VMWare’s default MAC address identifier groups are 00:0C:29. An easy check is to open a command prompt and type *getmac* and look for the VMWare MAC identifier as shown in Figure 9.



```
C:\Windows\system32>getmac

Physical Address    Transport Name
=================== ==========================================================
00-0C-29-88-04-5F   \Device\Tcpip_{84DBFF13-F274-4246-A556-71FD7ABBC7A9}

C:\Windows\system32>_
```

Figure 9. VMWare MAC Identifier 00-0C-29

(U) Using Sensitive Instructions to Detect VMWare

(U) Because not all instructions can be virtualized, VMWare uses what is known as binary translation for these 'problematic' instruction. Binary translation traps the problematic instructions, and essentially changes context to handle the instruction on the host processor, returning the result and resuming the virtualization. Naturally, this causes a significant performance hit and timing methods can be used to determine if the target is running in VMWare.

(U) Some instructions return different results if run in VMWare as compared to running on native hardware. These instructions can be used to determine if a VMWare machine is present.

(U) The Red Pill Anti-VM Technique

(U) The Red Pill anti-VM technique uses the sidt instruction to get the value of the IDTR register. The VM has to relocate the guest IDTR to avoid conflict with the host's IDTR. Because the VM is not notified when the VM runs the sidt instruction, the IDTR for the VM is returned. The fifth byte of the IDTR returned contains the start of the base memory address. VMWare returns 0xFF in the fifth byte. The Red Pill tests for this discrepancy to detect VMWare. It should be noted that this technique only works reliably on single processor machines (VMWare default is single processor).

(U) The No Pill Anti-VM Technique

(U) The No Pill anti-VM technique uses the sgdt and sldt and relies on the fact the Local Descriptor Table (LDT) structure is assigned to a processor, not the OS. Because Windows does not use the LDT structure but VMWare provides support for it the table will vary in predictable ways. The location of LDT on the host machine will be zero while the location of the LDT of a VMWare guest machine will be a non-zero value. A simple check for a non-zero return from either sgdt or sldt will indicate the OS is a VM image.

(U) Checking the I/O Communications Port

(U) VMWare uses a virtual I/O port with a specific 'magic number' for communications between the host machine and the guest machine. The port can be queried and compared with the magic number to identify VMWare. The magic number, in hex is 0x564D5868, converts to ASCII "VMXh".

(U) This technique relies on the x86 instruction *in*, which copies data from the I/O port specified by the source operand to a memory address specified by the destination operand. VMWare monitors the use of the *in* operator and captures the I/O destination for the communication port 0x5668 (VX). Therefore, the second operand needs to be loaded with VX in order to check for VMWare, which only happens when EAX holds the magic number 0x564D5868. ECX mustg contain a value corresponding to the action you want to perform on the port. The value 0xA means, "get the VMWare version type", and 0x14 means, "get the memory size." Either can be used to detect VMWare.

Timing-based VM Detection

(U) As mentioned earlier, virtualization environments cannot emulate every instruction. Some instructions are trapped in the kernel, the virtualization environment is halted, the instruction is handled by the host processor and the result passed back to the emulation environment and the emulation environment is restarted. Naturally, all this causes quite a performance hit and that degradation of performance can be measured via timestamps.

(U) One of the problematic instructions is CPUID. The timing-based VM detection technique involves taking a timestamp, looping a large number of sequential CPUID calls (> 4000), taking another timestamp and calculating the difference. If running in an emulation environment, the time difference will be orders of magnitude greater than the time difference when running on a native host system.

(U) Resources

www.symantec.com/connect/articles/windows-anti-debug-reference

Andrew Honig and Michael Sikorski, Practical Malware Analysis, No Starch Press 2012

http://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged

https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf

https://www.exploit-db.com/docs/34591.pdf