TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART EIGHT

Peter Ferrie
Microsoft, USA

New anti-unpacking tricks continue to be developed as older ones are constantly being defeated. Last year, a series of articles described some tricks that might become common in the future, along with some countermeasures [1–8]. Now, the series continues with a look at tricks that are specific to debuggers and emulators.

INTRODUCTION

Anti-unpacking tricks can come in different forms, depending on what kind of unpacker they are intended to attack. The unpacker can be in the form of a memory dumper, a debugger, an emulator, a code buffer, or a W-X interceptor. It may also be a tool in a virtual machine. There are corresponding tricks for each of these. This article and the ones that follow look at some of the tricks that are specific to debuggers and emulators. Definitions of these are as follows:

- A debugger attaches to a process, allowing single-stepping, or the placing of breakpoints at key locations, in order to stop execution at the right place. The process can then be dumped with more precision than using a memory dumper alone.
- An emulator, as referred to within this paper, is a purely software-based environment, most commonly used by anti-malware software. It places the suspicious file inside the environment and watches its execution for particular events of interest.

Unless stated otherwise, all of the techniques described in this article were discovered and developed by the author.

ANTI-UNPACKING BY ANTI-DEBUGGING

1. PEB fields

1.1 NtGlobalFlag

For a 32-bit process on a 64-bit platform, there are separate PEBs for the 32-bit portion and the 64-bit portion. The 64-bit PEB contains copies of the same interesting fields as the 32-bit PEB, though the locations are different between the two. As such, the NtGlobalFlag field exists at offset 0xbc in the 64-bit PEB. The value in that field is zero by default. As with 32-bit platforms, there is a particular value that is typically stored in the field when a debugger is running. The presence of that value is not a reliable indication that a debugger is running, but it could be used for that purpose. All of the information regarding this field is identical to the 32-bit version, and was described in [1]. However, there are no current that tools hide the 64-bit NtGlobalFlag flags.

Example 32-bit code to detect the 64-bit default value looks like this:

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

1.2 Heap flags

For a 32-bit process on a 64-bit platform, there are separate heaps for the 32-bit portion and the 64-bit portion. Within the 32-bit heap are two well-known fields of interest. The same fields exist in the 64-bit heap, with the same flags. As a result, the same vector exists for detecting a debugger. The PEB64->NtGlobalFlag field forms the basis for the values in those fields. No current tools hide the 64-bit heap flags.

Example 32-bit code to detect the 64-bit value looks like this:

```
mov eax, fs:[30h] ;PEB
;64-bit PEB follows 32-bit PEB
;get process heap base
mov eax, [eax+1030h]
cmp d [eax+70h], 2 ;Flags
jne being_debugged
```

and this:

```
mov eax, fs:[30h] ;PEB
;64-bit PEB follows 32-bit PEB
;get process heap base
mov eax, [eax+1030h]
cmp d [eax+74h], 0 ;ForceFlags
jne being_debugged
```

1.3 The heap

The problem with simply clearing the heap flags is that the initial heap will have been initialized with the flags active, and that leaves some artefacts that can be detected. Specifically, at the end of the heap block there will be one definite value, and one possible value. The HEAP_TAIL_CHECKING_ENABLED flag causes the sequence 0xABABABAB to appear four times at the exact end of the allocated block in the 64-bit heap. The HEAP_FREE_CHECKING_ENABLED flag causes the sequence 0xFEEEFEEE (or a part thereof) to appear if additional bytes are required to fill in the slack space until the next

block. *Windows Vista* strengthened the 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 it is used by default.

Example 32-bit code to detect the 32-bit value looks like this:

```
mov     eax, <heap ptr>
;get unused_bytes
movzx   edx, w [eax-8] ;size
xor     ebx, ebx
mov     ecx, fs:[ebx+30h] ;PEB
;get process heap base
mov     ecx, d [ecx+18h]
;check for protected heap
cmp     d [ecx+4ch], ebx
;get heap key
cmovne ebx, [ecx+50h]
xor     dx, bx
movzx   ecx, b [eax-1]
sub     eax, ecx
lea     edi, [edx*8+eax]
mov     al, 0abh
mov     cl, 8
repe    scasb
je      being_debugged
```

Example 32-bit code to detect the 64-bit value looks like this:

```
mov     eax, <heap ptr>
;get unused_bytes
movzx   edx, w [eax-8] ;size
xor     ebx, ebx
mov     ecx, fs:[ebx+30h] ;PEB
;64-bit PEB follows 32-bit PEB
;get process heap base
mov     ecx, [ecx+1030h]
;check for protected heap
cmp     d [ecx+7ch], ebx
;get heap key
cmovne ebx, [ecx+88h]
xor     dx, bx
add     edx, edx
movzx   ecx, b [eax-1]
sub     eax, ecx
lea     edi, [edx*8+eax]
mov     al, 0abh
mov     cl, 10h
repe    scasb
je      being_debugged
```

2. Special APIs

2.1 IsDebuggerPresent

The kernel32 IsDebuggerPresent() function simply returns the value of the PEB->BeingDebugged flag. However, the PEB64->BeingDebugged flag exists, and can be queried directly. The *Stealth64* plug-in for *OllyDbg* is currently the only tool that hides the 64-bit PEB->BeingDebugged flag.

Example code looks like this:

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

2.2 NtSetDebugFilterState

The ntdll NtSetDebugFilterState() function can be used to detect the presence of a debugger.

Example code looks like this:

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

However, the function requires the calling process to possess the debug privilege. Example code to acquire the debug privilege looks like this:

```
   xor  ebx, ebx
   push 2
   push ebx
   push ebx
   push esp
   push offset l1
   push ebx
   call LookupPrivilegeValueA
   push eax
   push esp
   push 20h
   push -1 ;GetCurrentProcess()
   call OpenProcessToken
   pop  ecx
   push eax
   mov  eax, esp
   push ebx
   push ebx
   push 10h
   push eax
   push ebx
   push ecx
   call AdjustTokenPrivileges
   ...
l1: db "SeDebugPrivilege", 0
```

This method has been disclosed publicly [9].

2.3 RtlQueryProcessDebugInformation

The ntdll RtlQueryProcessDebugInformation() function can be used to read certain fields indirectly from the process

memory of any given process. Specifically, the heap flags can be read using this function, and it is not obvious that it is being done. This method has been disclosed publicly [10]. However, while the basic idea is valid, it does not work as described because one of the flags was removed in *Windows Vista*. Thus, the flags value should be masked first.

Example correct code looks like this:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
;PDI_HEAPS + PDI_HEAP_BLOCKS
push  14h
call  GetCurrentProcessId
push  eax
call  RtlQueryProcessDebugInformation
;HeapInformation
mov   eax, [ebx+38h]
mov   eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
;(missing in Vista)
and   al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ CREATE_ALIGN_16
;+ VALIDATE_PARAMETERS_ENABLED
cmp   eax, 62000140h
je    being_debugged
```

However, it is better to compare with the value that is set when no debugger is present.

Example code looks like this:

```
push 0
push 0
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
;PDI_HEAPS + PDI_HEAP_BLOCKS
push 14h
call GetCurrentProcessId
push eax
call RtlQueryProcessDebugInformation
;HeapInformation
mov  eax, [ebx+38h]
;HEAP_GROWABLE
cmp  d [eax+8], 2 ;Flags
jne  being_debugged
```

2.4 RtlQueryProcessHeapInformation

The ntdll RtlQueryProcessHeapInformation() function can be used to read the heap flags indirectly from the process memory for the current process, and it is not obvious that it is being done. This method has also been disclosed publicly [11]. However, the disclosure refers to the wrong structure, so the description is incorrect. The accepted parameter is a pointer to a DEBUG_HEAP_INFORMATION structure, not a DEBUG_BUFFER structure. As a result, the DEBUG_BUFFER->RemoteSectionBase field in the text is actually the DEBUG_HEAP_INFORMATION->Flags field. Given that correction, it all makes sense, and we can see that the Flags check is a variation of the above method, but without the indirect pointer.

Example code looks like this:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
call RtlQueryProcessHeapInformation
mov   eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
;(missing in Vista)
and   al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ CREATE_ALIGN_16
;+ VALIDATE_PARAMETERS_ENABLED
cmp   eax, 62000140h
je    being_debugged
```

As before, it is better to compare with the value that is set when no debugger is present.

Example 'correct' code looks like this:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
call  RtlQueryProcessHeapInformation
;HEAP_GROWABLE
cmp   d [eax+8], 2 ;Flags
jne   being_debugged
```

However, there is an assumption in this code which has been shown to be invalid in *Windows Vista*. The assumption is that the debug heap information begins four bytes after the start of the debug buffer. This is true for platforms prior to *Windows Vista* because the DEBUG_BUFFER->SizeOfInfo field is not initialized. However, in *Windows Vista* this field is initialized to a non-zero value. As a result, the correct way of accessing the debug heap information is via the indirect pointer, as for the ntdll RtlQueryProcessDebugInformation() function method.

This becomes more obvious because the ntdll RtlQueryProcessDebugInformation() function calls the ntdll RtlQueryProcessHeapInformation() function internally, and passes the original buffer pointer. As a result, the method for accessing the contents should be the same in both cases.

Example correct code using the debugger value looks like this:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
call  RtlQueryProcessHeapInformation
;HeapInformation
mov   eax, [ebx+38h]
mov   eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
;(missing in Vista)
and   al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ CREATE_ALIGN_16
;+ VALIDATE_PARAMETERS_ENABLED
cmp   eax, 62000140h
je    being_debugged
```

Example correct code using the default value looks like this:

```
push  0
push  0
call  RtlCreateQueryDebugBuffer
push  eax
xchg  ebx, eax
call  RtlQueryProcessHeapInformation
;HeapInformation
mov   eax, [ebx+38h]
;HEAP_GROWABLE
cmp   d [eax+8], 2 ;Flags
jne   being_debugged
```

A variation of this technique which checks a different field has been disclosed publicly [12]. However, the text refers to the wrong structure, so the description is incorrect. As above, the accepted parameter is a pointer to a DEBUG_HEAP_INFORMATION structure, not a DEBUG_BUFFER structure. As a result, the DEBUG_BUFFER->RemoteSectionBase field in the text is actually the DEBUG_HEAP_INFORMATION->Flags field, and the DEBUG_BUFFER->EventPairHandle field is actually the DEBUG_HEAP_INFORMATION->Allocated field. As above, we can see that the Flags check is a variation of the above method, and that the Allocated check is an unreliable method.

2.5 CloseHandle

It is well known that the kernel32 CloseHandle() function and the ntdll NtClose() function will raise an exception if an invalid or protected handle is passed to the function in the presence of a debugger. However, it is less well known that there is a global flag that can be set to always produce this behaviour. Setting the value 0x400000 (FLG_ENABLE_CLOSE_EXCEPTIONS) in the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value, and then rebooting, causes the kernel32 CloseHandle() function and the ntdll NtClose() function to always raise an exception if an invalid or protected handle is passed to the function, even if no debugger is present. This behaviour is supported on all *NT*-based versions of *Windows*, on both the 32-bit and 64-bit platforms.

At the time of writing, *Microsoft* documentation claims that other APIs that receive handles (such as the kenel32 SetEvent() function) will behave in the same way when this flag is set [13], but this claim is incorrect. There are only two places in the kernel that raise a user-mode exception based on this flag, and both of them are in the ntoskrnl NtClose() function.

It has been claimed that the ntoskrnl NtClose() function is the only kernel-mode function that raises a user-mode exception [14]. This is also incorrect. The ntoskrnl ObReferenceObjectByHandle() function also raises a user-mode exception – but the circumstances for the exception are different. Setting the value 0x100 (FLG_APPLICATION_VERIFIER) in the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value, and then rebooting, causes the ntoskrnl ObReferenceObjectByHandle() function to 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. It seems likely that this flag is the one the author of the *Microsoft* documentation had in mind. The exception-raising behaviour of this flag is not documented.

There is another flag – 0x40000000 (FLG_ENABLE_HANDLE_EXCEPTIONS) – which, at the time of writing, *Microsoft* documentation claims will raise a user-mode exception when an invalid handle is passed to the Object Manager [15]. However, this claim is also incorrect. While an exception is raised in response to an invalid handle, the handle is accepted only from kernel mode, not from user mode. The exception occurs in kernel mode (specifically, a bug check event and a blue screen), the exception is not

passed to user mode, and this behaviour applies only to drivers. This flag was introduced in *Windows XP*.

Some third-party websites state that the FLG_ENABLE_CLOSE_EXCEPTIONS behaviour can be set on a per-process basis, but this is incorrect. The effect is system wide.

In *Windows Vista* and later versions, a further location was added that can raise user-mode exceptions, but it is reached only if an exception can be generated in kernel mode.

3. Process tricks

3.1 Thread local storage (TLS)

'Does my TLS callback run on attach?' is a simple question with a complex answer. When a process starts, the ntdll LdrInitializeThunk() function processes the PEB->Ldr->InLoadOrderModuleList list. The PEB->Ldr->InLoadOrderModuleList list contains the names of DLLs to process. The PLDR_DATA_TABLE_ENTRY->Flags value 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 only from ntdll.dll will not have Thread Local Storage callbacks executed on attach. *Windows 2000* and earlier contained a bug causing it to crash 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 load kernel32.dll explicitly, before processing the host import table. When kernel32.dll is loaded, it is added to the PEB->Ldr->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 in this particular case it is triggered 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 only from 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, provided that the DLL has a non-zero entrypoint. Then the TLS callbacks will be executed on attach. The workaround is effective because the PLDR_DATA_TABLE_ENTRY->Flags value will have the LDRP_ENTRY_PROCESSED bit clear for that DLL.

This problem has been known about since at least 2005 [16], and has been described in part [17–19], but the exact cause has never been disclosed until now.

This behaviour could be an effective anti-emulation trick for a file that imports only from kernel32.dll or ntdll.dll. It would detect emulators that always run the Thread Local Storage callbacks by default.

Example code looks like this:

```
   mov  ecx, d [esp+8] ;reason
   loop l1 ;not DLL_PROCESS_ATTACH
   call GetVersion
   cmp  al, 5
   jnbe being_emulated ;Vista+
   jb   l1
   test ah, ah
   jne  being_emulated ;XP or later
l1: ...
```

3.2 Import table

Windows trims spaces and periods while processing the module names in an import table before attempting to load the file. The kernel32 LoadLibrary() function behaves in the same way. Thus, 'kernel32.dll' is almost equivalent to 'kernel32.dll.' or 'kernel32.dll. . . . .'.

The caveat here is that *Windows* checks if a module is loaded already by examining the original name, not the normalized one. As a result, if spaces and/or periods are appended to the string, then a new copy of the DLL will be loaded.

In contrast, the kernel32 GetModuleHandle() function will remove only one period and no spaces. For example, calling the kernel32 GetModuleHandle('kernel32.dll.') function will return the address of the 'real' kernel32.dll, not the one with the appended period. Thus, if importing from 'kernel32.dll.', and then requesting the module handle of 'kernel32.dll.', the values will be different.

Example code looks like this:

```
    push offset l1
    mov  eax,
           [offset GetModuleHandleA+2]
    mov  ebx, [eax]
    call ebx
    cmp  eax, ebx
    jnb  being_debugged
    ...
l1: db "kernel32.dll.", 0
```

Further, calling the kernel32 GetModuleHandle() function will fail for a DLL which was loaded using appended characters. Thus, loading 'kernel32.dll..' should succeed, but requesting the module handle of 'kernel32.dll..' should fail.

Example code looks like this:

```
    mov  esi, offset l1
    push esi
    call LoadLibraryA
    test eax, eax
    je   being_debugged
    push esi
    call GetModuleHandleA
    test eax, eax
    jne  being_debugged
    ...
l1: db  "kernel32.dll..", 0
```

The next part of this series will continue to look at anti-debugging tricks, including looking at self-modifying code, selectors, RDTSC and *Syser* plug-ins.

The text of this paper was produced without reference to any Microsoft source code or personnel.

REFERENCES

[1] Ferrie, P. Anti-unpacker tricks. http://pferrie.tripod.com/papers/unpackers.pdf.

[2] Ferrie, P. Anti-unpacker tricks – part one. Virus Bulletin, December 2008, p.4. http://www.virusbtn.com/pdf/magazine/2008/200812.pdf.

[3] Ferrie, P. Anti-unpacker tricks – part two. Virus Bulletin, January 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200901.pdf.

[4] Ferrie, P. Anti-unpacker tricks – part three. Virus Bulletin, February 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200902.pdf.

[5] Ferrie, P. Anti-unpacker tricks – part four. Virus Bulletin, March 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200903.pdf.

[6] Ferrie, P. Anti-unpacker tricks – part five. Virus Bulletin, April 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200904.pdf.

[7] Ferrie, P. Anti-unpacker tricks – part six. Virus Bulletin, May 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200905.pdf.

[8] Ferrie, P. Anti-unpacker tricks – part seven. Virus Bulletin, June 2009, p.4 http://www.virusbtn.com/pdf/magazine/2009/200906.pdf.

[9] Giuseppe 'Evilcry' Bonfa'. NtSetDebugFilterState as Anti-Dbg Trick Reverse Engineering. http://evilcry.netsons.org/tuts/NtSetDebugFilterState.pdf.

[10] RtlQueryProcessDebugInformation as Anti-Dbg Trick. http://evilcodecave.wordpress.com/2009/04/11/rtlqueryprocessdebuginformation-as-anti-dbg-trick/.

[11] RtlQueryProcessHeapInformation As Anti-Dbg Trick. http://evilcodecave.wordpress.com/2009/04/14/rtlqueryprocessheapinformation-as-anti-dbg-trick/.

[12] EventPairs Reversing – EventPairHandle as Anti-Dbg Trick. http://evilcodecave.wordpress.com/2009/05/06/eventpairs-reversing-eventpairhandle-as-anti-dbg-trick/.

[13] MSDN Library. Enable close exception. http://msdn.microsoft.com/en-us/library/ff542887.aspx.

[14] Newger, J. New IDA Stealth – Improved anti-anti-debugging techniques. http://newgre.net/node/43.

[15] MSDN Library. Enable bad handles detection. http://msdn.microsoft.com/en-us/library/ff542881.aspx.

[16] User32.dll init in Longhorn. http://blogs.msdn.com/mgrier/archive/2005/06/24/432455.aspx#433355.

[17] XP/S2K3 fails to process TLS w/o USER32. http://nezumi-lab.org/blog/?p=15.

[18] TLS callbacks w/o USER32 (part II). http://nezumi-lab.org/blog/?p=43.

[19] TLS callbacks w/o USER32 (part III). http://nezumi-lab.org/blog/?p=51.