TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART TWO

Peter Ferrie
Microsoft, USA

In the first part of this series last month (see *VB*, December 2008, p.4) we looked at a number of anti-unpacking tricks that have come to light recently. New anti-unpacking tricks continue to be developed because the older ones are constantly being defeated. In this article and the ones that follow, we will describe some tricks that might become common in the future, along with some countermeasures.

INTRODUCTION

Anti-unpacking tricks 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 can also be a tool in a virtual machine. There are corresponding tricks for each of these.

- A *memory-dumper* dumps the process memory of the running process without regard to the code inside it.
- A *debugger* attaches to the 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 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 file to execute inside the environment and watches the execution for particular events of interest.
- A *code-buffer* is similar to a debugger. It also attaches to a process, but instead of executing instructions in place, it copies each instruction into a private buffer and executes it from there. It allows fine-grained control over execution as a result. It is also more transparent than a debugger, and faster than an emulator.
- A *W-X interceptor* uses page-level tricks to watch for write-then-execute sequences. Typically, an executable region is marked as read-only and executable, and then everything else is marked as read-only and non-executable (or simply non-present, depending on the hardware capabilities). Then the code is allowed to execute freely. The interceptor intercepts exceptions that are triggered by writes to read-only pages, or execution from non-executable or non-present pages. If the hardware supports it, a read-only page will be replaced by a writable but non-executable page, and then the write will be allowed to continue. Otherwise, the single-step exception will be used to allow the write to complete, after which the page will be restored to its non-present state. In either case, the page address is kept in a list. In the event of exceptions triggered by execution of non-executable or non-present pages, the page address is compared to the entries in that list. A match indicates the execution of newly written code, and is a possible host entry point.

Now we move to potentially new tricks. All of these techniques were discovered and developed by the author of this paper. This article will concentrate on anti-debugging tricks.

ANTI-UNPACKING BY ANTI-DEBUGGING

1. Heap flags

Within the heap are two fields of interest. The PEB->NtGlobalFlag field forms the basis for the values in those fields. It should be noted that the HEAP_VALIDATE_PARAMETERS_ENABLED flag value was changed in *Windows XP* and later, from 0x200000 to 0x40000000, and that a new NtGlobalFlag flag 0x80 (FLG_HEAP_VALIDATE_ALL) was introduced (which corresponds to the HEAP_VALIDATE_ALL_ENABLED flag). Further, the location of the Flags and ForceFlags fields is different in *Windows Vista*. No current packer supports the new location, which is the reason why some packers will not run on *Windows Vista*.

Example code for *Windows Vista* looks like this:

```
mov eax, fs:[30h] ;PEB
;get process heap base
mov eax, [eax+18h]
mov eax, [eax+40h] ;Flags
dec eax
dec eax
jne being_debugged
```

and this:

```
mov eax, fs:[30h] ;PEB
;get process heap base
mov eax, [eax+18h]
cmp d [eax+44h], 0 ;ForceFlags
jne being_debugged
```

2. Special APIs

2.1 CreateFile

The kernel32 CreateFile() function can be used to open a file for exclusive access. This technique is not new in general, but it is new with respect to debugger detection techniques.

Example code looks like this:

```
    xor   ebx, ebx
    mov   ebp, offset l1
    push  104h ;MAX_PATH
    push  ebp
    push  ebx ;self filename
    call  GetModuleFileNameA
    push  ebx
    push  ebx
    push  3 ;OPEN_EXISTING
    push  ebx
    push  ebx
    push  80000000h ;GENERIC_READ
    push  ebp
    call  CreateFileA
    inc   eax
    je    being_debugged
    ...
l1: db    104h dup (?) ;MAX_PATH
```

This technique works against the debugger *Turbo Debug32*, but not debuggers such as *OllyDbg* and *WinDbg*. It is related to the debug privilege, which debuggers such as *OllyDbg* and *WinDbg* maintain, while *Turbo Debug32* does not.

2.2 RaiseException

The kernel32 RaiseException() function can be used to force certain exceptions to occur. These include exceptions that a debugger would normally consume.

Turbo Debug32 consumes the following exceptions:

```
0x40010005 (DBG_CONTROL_C)
0x40010007 (DBG_RIPEVENT)
0x80000002 (DATATYPE_MISALIGNMENT)
0x80000003 (BREAKPOINT)
0x80000004 (SINGLE_STEP)
0x80000029 (UNWIND_CONSOLIDATE)
0xC0000005 (ACCESS_VIOLATION)
0xC000008C (ARRAY_BOUNDS_EXCEEDED)
0xC000008D (FLOAT_DENORMAL_OPERAND)
0xC000008E (FLOAT_DIVIDE_BY_ZERO)
0xC000008F (FLOAT_INEXACT_RESULT)
0xC0000090 (FLOAT_INVALID_OPER)
0xC0000091 (FLOAT_OVERFLOW)
0xC0000092 (FLOAT_STACK_CHECK)
0xC0000093 (FLOAT_UNDERFLOW)
0xC0000094 (INTEGER_DIVIDE_BY_ZERO)
0xC0000095 (INTEGER_OVERFLOW)
0xC0000096 (PRIVILEGED_INSTRUCTION)
```

When raised in the presence of *Turbo Debug32,* none of these exceptions will be delivered to the debuggee. The missing exception can be used to infer the presence of *Turbo Debug32*.

Example code looks like this:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    push eax
    push eax
    push eax
    ;DBG_CONTROL_C
    push 40010005h
    call RaiseException
    jmp  being_debugged
l1: ...
```

By default, *OllyDbg* will consume a similar list of exceptions, but it can be configured to pass them to the debuggee.

The *Interactive DisAssembler* (*IDA*) debugger consumes the following exceptions:

```
0x40010006 (DBG_PRINTEXCEPTION_C)
0x40010007 (DBG_RIPEVENT)
0x80000003 (BREAKPOINT)
```

It is known that *WinDbg* consumes the DBG_PRINTEXCEPTION_C (0x40010006) exception, though this fact is used only rarely. However, *WinDbg* also consumes the following exceptions:

```
0x40000005 (SEGMENT_NOTIFICATION)
0x40010005 (DBG_CONTROL_C)
0x40010007 (DBG_RIPEVENT)
0x40010008 (DBG_CONTROL_BREAK)
0x40010009 (DBG_COMMAND_EXCEPTION)
0x80000001 (GUARD_PAGE_VIOLATION)
0xC0000420 (ASSERTION_FAILURE)
```

The SEGMENT_NOTIFICATION (0x40000005) exception is of particular interest, since it can be used to demonstrate several behaviours. One of these behaviours is to force a break into the VDM debugger prompt.

Example code looks like this:

```
    push offset l1
    push 4
    push 0
    ;EXCEPTION_SEGMENT_NOTIFICATION
    push 40000005h
    call RaiseException
    ...
l1: dd  0c0000002h, 0
    dd  offset l1, offset l1
    dd  0, 0, offset l1
    db  2b0h dup (0)
```

Another of the behaviours is to cause the debugger to remove a breakpoint from the specified location in the debuggee's process memory.

Example code looks like this:

```
    push offset l4
    push 4
    push 0
    ;EXCEPTION_SEGMENT_NOTIFICATION
    push 40000005h
    call RaiseException
    push offset l5
    push 1
```

```
      push 0
      ;EXCEPTION_SEGMENT_NOTIFICATION
      push 40000005h
      ;remove breakpoint
      call RaiseException
  l1: mov al, 0cch
      ...
  l2: dd  0
  l3: dd  offset l7
  l4: dd  2 ;dummy context request
  l5: dd  6, offset l2, offset l3
      dd  0, offset l2
  l6: db  3, 90h ;replacement value
      db  0ah dup (0)
  l7: dw  0
      db  offset l1 + 1
      db  (offset l1 + 1) shr 8
      db  (offset l1 + 1) shr 10h
      dw  0
      db  (offset l1 + 1) shr 18h
      dd  0, offset l6
      db  7ch dup (0)
      dw  1
      db  8 dup (0), 1, 209h dup (0)
```

In this case, the value in AL at l1 is altered from 0xCC to 0x90.

In *Windows Vista*, there are two new exceptions. They are EXCEPTION_WX86_SINGLE_STEP (0x4000001E) and EXCEPTION_WX86_BREAKPOINT (0x4000001F). As their names imply, they are the x86 equivalents of EXCEPTION_BREAKPOINT (0x80000003) and EXCEPTION_SINGLE_STEP (0x80000004). When a single-step or breakpoint occurs in 32-bit mode, these new exceptions are raised instead of the old ones. If a debugger does not handle them, then the kernel translates them to the old values and dispatches them again. In either case, they will be consumed by the debugger if that was the previous behaviour.

2.3 DbgBreakPoint

The ntdll DbgBreakPoint() function is called when a debugger attaches to a process that is already running. This allows the debugger to gain control because an exception is raised that it can intercept. This technique can be defeated simply by erasing the breakpoint.

Example code looks like this:

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

If a debugger attempts to attach to a process that contains such a change, then the thread will exit immediately, and the debugger will not break in. *Turbo Debug32*, and possibly other console-mode debuggers, will hang as a result, because they wait infinitely for an exception to be raised in order to continue execution.

2.4 OutputDebugString

Despite the fact that the kernel32 OutputDebugString() function raises the DBG_PRINTEXCEPTION_C (0x40010006) exception, a registered Structured Exception Handler will not see it. The reason is that *Windows* registers its own Structured Exception Handler internally, which consumes the exception if a debugger does not do so. As such, the presence of a debugger that consumes the exception cannot be inferred by the absence of the exception.

However, in *Windows XP* and later, any registered Vectored Exception Handler will run before the Structured Exception Handler that *Windows* registers. This might be considered a bug in *Windows*. In this case the presence of a debugger that consumes the exception can be inferred by its absence.

2.5 DbgPrint

Similarly, despite the fact that the ntdll DbgPrint() function raises the DBG_PRINTEXCEPTION_C (0x40010006) exception, a registered Structured Exception Handler will not see it. Once again, the reason is that *Windows* registers its own Structured Exception Handler internally, which consumes the exception if a debugger does not do so. As such, the presence of a debugger that consumes the exception cannot be inferred by the absence of it.

However, as discussed previously, in *Windows XP* and later, any registered Vectored Exception Handler will run before the Structured Exception Handler that *Windows* registers and the presence of a debugger that consumes the exception can now be inferred by the absence of the exception. Further, a different exception is delivered to the Vectored Exception Handler if a debugger is present but has not consumed the exception, or if a debugger is not present. If a debugger is present but has not consumed the exception, then *Windows* will deliver the DBG_PRINTEXCEPTION_C (0x40010006) exception. If a debugger is not present, then *Windows* will deliver the EXCEPTION_ACCESS_VIOLATION (0xC0000005) exception. The presence of a debugger can now be inferred by either the absence of the exception, or by the value of the exception.

2.6 LoadLibrary

The kernel32 LoadLibrary() function is an unexpected method for debugger detection, but a simple and effective one. When a file is loaded in the presence of a debugger using the kernel32 LoadLibrary() function, and then freed, a handle remains open for that file. As a result, the file can no longer be opened for exclusive access. This fact can be used to infer the presence of the debugger.

Example code looks like this:

```
     mov   esi, offset l1
     push esi
     call LoadLibraryA
     push eax
     call FreeLibrary
     xor   ebx, ebx
     push ebx
     push ebx
     push 3
     push ebx
     push ebx
     push 80000000h
     push esi
     call CreateFileA
     inc   eax
     je    being_debugged
     ...
l1:  db    "myfile", 0
```

A less obvious method of achieving the same thing is to use the resource-updating APIs, specifically the kernel32 EndUpdateResource() function. The reason this works is because it eventually calls the kernel32 CreateFile() function to write the new resource table.

Example code looks like this:

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

2.7 NtQueryInformationProcess

As with the ProcessDebugPort class mentioned in [1], two other classes are similarly affected by arbitrary patching without checking the process handle: ProcessDebugObjectHandle and ProcessDebugFlags.

Example code for the ProcessDebugObjectHandle class looks like this:

```
     xor   ebx, ebx
     mov   ebp, offset l1
     push ebp
     call GetStartupInfoA
     ;sizeof(PROCESS_INFORMATION)
     sub   esp, 10h
     push esp
     push ebp
     push ebx
     push ebx
     push 1 ;DEBUG_PROCESS
     push ebx
     push ebx
     push ebx
     push ebx
     push offset l2
     call CreateProcessA
     pop   eax
     push eax
     mov   ecx, esp
     push 0
     push 4 ;ProcessInformationLength
     push ecx
     ;ProcessDebugObjectHandle
     push 1eh
     push eax
     call NtQueryInformationProcess
     pop   eax
     test eax, eax
     je    being_faked
     ...
     ;sizeof(STARTUPINFO)
l1:  db    44h dup (?)
l2:  db    "myfile", 0
```

Example code for the ProcessDebugFlags class looks like this:

```
     xor   ebx, ebx
     mov   ebp, offset l1
     push ebp
     call GetStartupInfoA
     ;sizeof(PROCESS_INFORMATION)
     sub   esp, 10h
     push esp
     push ebp
     push ebx
     push ebx
     push 1 ;DEBUG_PROCESS
     push ebx
     push ebx
     push ebx
     push ebx
     push offset l2
     call CreateProcessA
     pop   eax
     push eax
     mov   ecx, esp
     push 0
     push 4 ;ProcessInformationLength
     push ecx
     push 1fh ;ProcessDebugFlags
     push eax
     call NtQueryInformationProcess
     pop  eax
     test eax, eax
     jne  being_faked
     ...
     ;sizeof(STARTUPINFO)
 l1: db   44h dup (?)
 l2: db   "myfile", 0
```

3. Hardware tricks

3.1 Execution timing

When a debugger is used to single-step through code, there is a significant delay between the execution of the individual instructions when compared to native execution. This delay can be measured using one of several possible time sources. These sources include the kernel32 QueryPerformanceCounter(), kernel32 GetSystemTime() and kernel32 GetLocalTime() functions, the winmm timeGetSystemTime() function, and interrupt 0x2A (also known as the KiGetTickCount() function).

4. Process Tricks

4.1 No import table

Windows NT and *Windows 2000* assume that an executable file contains an import table, and that as a result, kernel32.dll is loaded. Kernel32.dll can be loaded by importing a function directly from kernel32.dll, but it is also acceptable to import a function from another DLL that also imports from kernel32.dll (user32.dll, gdi32.dll, etc.).

Normally, if kernel32.dll is not present, a fault will occur at the location at which the context EIP points, because no page is mapped there. However, it is possible to change the value in the PE->ImageBase field to place the executable file in that location. Then, whenever the file is executed, it will receive control instead of causing a fault. Further, since ntdll.dll is always loaded, it is possible to make use of some of its functions, such as ntdll LdrLoadDll() and ntdll LdrGetProcedureAddress(), to resolve the required functions and execute normally.

4.2 Anti-debugging DLLs

Dynamically loaded DLLs are called initially with the DLL_PROCESS_ATTACH parameter. If they refuse to load, they will be called immediately again, but with the DLL_PROCESS_DETACH parameter. Statically loaded DLLs are also called with the DLL_PROCESS_ATTACH parameter. However, if they refuse to load, then the ntdll NtRaiseHardError() function will be called in order to display the message: 'The application failed to initialize properly'. Following that, the ntdll RtlRaiseStatus() function will be called.

In the absence of a debugger, this function will trigger an exception that cannot normally be intercepted, because all registered Structured Exception Handlers will have been removed already. However, if the topmost Structured Exception Handler is replaced, then it will be called via the ntdll RtlRaiseStatus() function call. This can allow a DLL to continue execution after a message that suggests that it terminated.

Example code looks like this:

```
     push esi
     xor  esi, esi
     fs:lodsd
     inc  eax
 l1: dec  eax
     xchg eax, esi
     lodsd
     inc  eax
     jnz  l1
     mov  d [esi], offset l2
     pop  esi
     ret
 l2: ...
```

In this case, l2 will gain control after the message box is dismissed.

4.3 TLS Callback

Thread Local Storage (TLS) callback is an old technique that remains relatively under-investigated. The following are some new extensions:

- The TLS callback array can be altered (later entries can be modified) and/or extended (new entries can be appended) at runtime. Newly added or modified callbacks will be called using the new addresses. There is no limit to the number of callbacks that can be placed. This technique has been disclosed publicly [2].

  Example callback code looks like this:

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

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

- TLS callback addresses can point outside of the image – for example, to newly loaded DLLs.

  Example callback code looks like this:

  ```
  l1: push offset l2
      call LoadLibraryA
      mov     [offset cbEnd], eax
      ret
  l2: db      "tls2", 0
  ```

In this case, the 'MZ' header of tls2.dll will be executed when the callback at l1 returns. The file header can be made executable despite DEP, using the SectionAlignment trick described in [3]. This allows the code to run without error.

- TLS callback addresses can contain RVAs of imported addresses from other DLLs if the import address table is altered to point into the callback array. Imports are resolved before callbacks are called, so imported functions will be called normally when the callback array entry is reached.
- TLS callbacks receive three stack parameters, which can be passed directly to APIs. The first parameter is the ImageBase of the host process. It could be used by APIs such as the kernel32 LoadLibrary() or kernel32 WinExec() functions. The ImageBase parameter will be interpreted by the kernel32 LoadLibrary() or kernel32 WinExec() functions as a pointer to the filename to load or execute. By creating a file called 'MZ[some string]', where 'some string' matches the host file header contents, the TLS callback will access the file without any explicit reference. Of course, the 'MZ' portion of the string can also be replaced manually at runtime, but many APIs rely on this signature, so the results of such a change are unpredictable.
- TLS callbacks are called whenever a thread is created or destroyed (unless the process calls the kernel32 DisableThreadLibraryCalls() or the ntdll LdrDisableThreadCalloutsForDll() functions). This includes the thread that is created by *Windows* when a debugger attaches to a process. The debugger thread is special in that its entrypoint does not point inside the image. Instead, it points inside kernel32.dll. Thus, a simple debugger detection method is to use a TLS callback to query the start address of each thread that is created.

  Example callback code looks like this:

```
   push eax
   mov  eax, esp
   push 0
   push 4
   push eax
   ;ThreadQuerySetWin32StartAddress
   push 9
   push -2 ;GetCurrentThread()
   call NtQueryInformationThread
   pop  eax
   cmp  eax, offset l1
   jnb  being_debugged
   ...
l1: <code end>
```

- Since TLS callbacks run before a debugger can gain control, the callback can make other changes, such as removing the breakpoint that is typically placed at the host entrypoint. When combined with the ntdll DbgBreakPoint() function patch, the result is a file that cannot be debugged by ordinary means. The debugger will attach to the debuggee, and then wait for the exception which will never occur. Using Ctrl-C to break in will work well enough to look at the code, but breakpoints that are placed within the other threads will not activate.

  Example callback code looks like this:

```
   push offset l2
   call GetModuleHandleA
   push offset l3
   push eax
   call GetProcAddress
   push eax
   push esp
   push 40h  ;PAGE_EXECUTE_READWRITE
   push 1
   push eax
   xchg ebx, eax
   call VirtualProtect
   mov  b [ebx], 0c3h
   ;<val> is byte at l1
   mov  b [offset l1], <val>
   pop  eax
   ret
l1: <host entrypoint>
   ...
l2: db  "ntdll", 0
l3: db  "DbgBreakPoint", 0
```

  Currently, it seems that no debugger handles this case. However, the fix is very simple, and increasingly necessary. It is a matter of inserting the breakpoint on the first byte of the first TLS callback instead of the host entrypoint. This will allow an exception to be raised as usual. However, care must be taken regarding the callback address, since as noted above, the address may be the RVA of an imported function. Thus, the address cannot be taken from the file header. It must be read from the image memory.

In part three of this article next month we will look at some miscellaneous anti-debugging tricks, as well as a range of tricks that target specific debuggers.

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

REFERENCES

[1] Ferrie, P. Anti-unpacker tricks – part one. Virus Bulletin, December 2008, p.4.

[2] Self-modifying TLS callbacks. http://www.openrce.org/blog/view/1114/Self-modifying_TLS_callbacks.

[3] Ferrie, P. Anti-unpacker tricks. 2008. http://pferrie.tripod.com/papers/unpackers.pdf.