

## ANTI-UNPACKER TRICKS – PART THIRTEEN

Peter Ferrie  
Microsoft, USA

New anti-unpacking tricks continue to be developed as older ones are constantly being defeated. This series of articles describes some tricks that might become common in the future, along with some countermeasures [1–13].

In this article we look at some anti-unpacking tricks that are specific to the *IDA* plug-in *IDA Stealth*.

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

### IDA PLUG-INS

A number of packers have been written to detect the *IDA* debugger, so one plug-in (so far) has been written to attempt to hide it from those packers. The following is a description of that plug-in, with a list of vulnerabilities that could be used to detect it.

### IDA STEALTH

The *IDA Stealth* plug-in was described in a previous paper [8]. What follows is a description of the changes from the previous version, and behaviour that is specific to more recent versions of *Windows*.

*IDA Stealth* sets the `PEB->Heap->ForceFlags` flags to zero, and clears all but the `HEAP_GROWABLE` flag in the `PEB->Heap->Flags` flags. However, the location of these fields was moved in *Windows Vista*, so the plug-in fails to hide the changes. *IDA Stealth* also clears all but the `FLG_STOP_ON_EXCEPTION`, `FLG_SHOW_LDR_SNAPS`, `FLG_DEBUG_INITIAL_COMMAND`, `FLG_STOP_ON_HUNG_GUI` and `FLG_HEAP_VALIDATE_ALL` bits in the `PEB->NtGlobalFlag` field. Whilst not as wrong as setting bits arbitrarily, this behaviour is still incorrect.

*IDA Stealth* patches the debugger's `ntdll RtlGetNtGlobalFlags()` function code to always return zero.

*IDA Stealth* hooks the debugger's `ntdll NtQuerySystemInformation()` function by replacing its first five bytes with a relative jump to an injected DLL. The hook intercepts any attempt to call the `ntdll NtQuerySystemInformation()` function with the `SystemKernelDebuggerInformation` class. When that occurs, the hook calls the original `ntdll NtQuerySystemInformation()` function, and exits if an error occurs. If no error occurs, then *IDA Stealth* will store a

value that corresponds to the cleared `KdDebuggerEnabled` flag and the `KdDebuggerNotPresent` flag that is set. However, it is unclear why this function is intercepted, since *IDA* is not a kernel-mode debugger.

The hook also checks if the `ntdll NtQuerySystemInformation()` function was called with the `SystemProcessInformation` class. If so, then the hook calls the original `ntdll NtQuerySystemInformation()` function. If the call is successful, and the 'hide IDA' option is enabled, then the hook searches within the returned buffer for 'idag.exe' or 'idaw.exe', then erases all copies of the name that are found.

If the 'fake parent' option is enabled, then the hook replaces the process ID of the *IDA* debugger with the process ID of `EXPLORER.EXE` in the `InheritedFromUniqueProcessId` field. This could be considered a bug, since the true parent might not be *Explorer*. The proper behaviour would be to use the process ID of *IDA*'s parent.

*IDA Stealth* also hooks the debugger's `ntdll NtQueryInformationProcess()` function by replacing its first five bytes with a relative jump to an injected DLL. The hook intercepts any attempt to call the `ntdll NtQueryInformationProcess()` function with the `ProcessDebugPort` class, and returns a zero for the debug port if the current process ID matches the requested process ID.

The hook also checks whether the `ntdll NtQueryInformationProcess()` function was called with the `ProcessBasicInformation` class, and that the current process ID matches the requested process ID. If both of those conditions are true, then the hook replaces the parent process ID with that of the shell window in the `InheritedFromUniqueProcessId` field. This could be considered a bug, since the true parent might not be the shell. The proper behaviour would be to use the process ID of *IDA*'s parent.

*IDA Stealth* hooks the debugger's `ntdll NtQueryObject()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. The hook intercepts attempts to call the `ntdll NtQueryObject()` function with the `ObjectAllTypesInformation` class. When that occurs, the hook calls the original `ntdll NtQueryObject()` function, then searches within the returned buffer for the 'DebugObject' string. The hook sets the object count to zero if the `DebugObject` is found.

The plug-in no longer hooks the debugger's `ntdll NtClose()` function. Instead, it patches the `ntdll KiUserExceptionDispatcher()` function by replacing the function's first byte with a 'C3' opcode ('RET' instruction). This has the effect of disabling the exception that is raised

when an invalid handle is passed to the ntdll NtClose() function. However, this technique only works on the 32-bit versions of *Windows*. On 64-bit versions of *Windows*, the ntdll KiRaiseExceptionDispatcher() function is called as before, but it is the 64-bit version of the function that is being called. That function calls the 64-bit kernel32 RaiseException() function, which eventually delivers the exception to the environment. As a result, the exception remains visible.

Apparently, it would be possible to apply the same first-byte replacement on the 64-bit platform, but it would require the use of an interesting trick. Specifically, the code must use a gate to enter 64-bit mode. From there, it would be a simple matter to call the ntdll NtProtectVirtualMemory() function to unprotect the memory, write the opcode as usual, and then call the ntdll NtProtectVirtualMemory() function again to re-protect the memory. Finally, the code must return to 32-bit mode through the gate. The existence of such a gate has been disclosed publicly [14]. This gate has some interesting properties. For example, it is impossible to query its attributes from user mode. The problem is that the 32-bit ntdll NtQueryInformationThread() function supports only three selectors for a selector query: 0x20 (which corresponds to the CS selector), 0x28 (which corresponds to the DS/ES/FS/GS selectors), and 0x50 (which corresponds to the FS selector). The results are also hard coded, and the behaviour is contained entirely within the wow64.dll file. The call never reaches ntoskrnl.exe.

In any case, disabling the exception without reference to the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value means that the absence of the exception might reveal the presence of *IDA Stealth*.

*IDA Stealth* hooks the debuggee's ntdll NtSetInformationThread() function by replacing its first five bytes with a relative jump to an injected DLL. The hook intercepts attempts to call the ntdll NtSetInformationThread() function with the HideThreadFromDebugger class. The hook detects an invalid handle by attempting to duplicate the handle. If the handle is valid, then the hook ignores the request and returns successfully.

The plug-in hooks the debuggee's kernel32 SuspendThread() function by replacing its first five bytes with a relative jump to an injected DLL. The hook detects an invalid handle by attempting to duplicate it. If the handle is valid, then the hook ignores the request and returns successfully.

*IDA Stealth* hooks the debuggee's kernel32 GetTickCount() function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached for the first time, it calls the kernel32

QueryPerformanceCounter() function to get an initial value for the tick count. Subsequent calls to the hook cause it to return a tick count that increases in a non-linear fashion.

The plug-in hooks the debuggee's user32 BlockInput() function by replacing its first five bytes with a relative jump to an injected DLL. When the hook is reached, it simply returns successfully. However, this behaviour is incorrect. *Windows* will not allow the input to be blocked twice, nor will it allow the input to be enabled twice. Thus, if the same state is passed to the function twice, the result should be different.

Example code looks like this:

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

*IDA Stealth* hooks the debuggee's kernel32 OpenProcess() function once again by replacing its first five bytes with a relative jump to an injected DLL. When the hook is reached, it enumerates the list of processes in order to find the CSRSS.EXE process. If that process is found, then its process ID is compared to the requested process ID. If there is a match, then the hook returns an error code. Otherwise, it calls the original function.

*IDA Stealth* hooks the debuggee's user32 SwitchDesktop() function by replacing the first five bytes of the function with a relative jump to an injected DLL. The hook detects an invalid handle by attempting to duplicate the handle. If the handle is valid, then the hook ignores the request and returns successfully.

The plug-in hooks the debugger's ntdll DbgUiConvertStateChangeStructure() function, if it is available (the API was introduced in *Windows XP*), by replacing its first five bytes with a relative jump to the plug-in. When the hook is reached, it checks for the DBG\_PRINTEXCEPTION\_C (0x40010006) exception, and then simply returns success if it is seen. Otherwise, it calls the original function. This allows the exception to be delivered to the debuggee.

*IDA Stealth* hooks the debuggee's kernel32 GetThreadContext() function by replacing its first five bytes with a relative jump to an injected DLL. When the hook is reached, it calls the original kernel32 GetThreadContext() function, and then either zeroes the contents of the debug registers, or returns a cached copy of the debug registers. There is a bug in this code which is that the register cache is identified by the current thread ID, instead of the ID of the thread for which the context is being retrieved. Thus, if

multiple threads request the same information for the same thread, they might receive different results.

*IDA Stealth* hooks the debuggee's kernel32

SetThreadContext() function by replacing its first five bytes with a relative jump to an injected DLL. When the hook is reached, it caches a copy of the debug registers, and then removes the CONTEXT\_DEBUG\_REGISTERS flag from the ContextFlags field, before completing the call. There are two bugs in that code. The first is that the register cache is identified by the current thread ID, instead of the ID of the thread for which the context is being set. Thus, if one thread sets the context for a second thread, and a third thread retrieves the context for the second thread, then the context might be different. The second bug is that the hook does not check if the lpContext parameter points to a valid memory address, that the lpContext range is readable, or that the first four bytes of the lpContext range are writable. If the lpContext pointer is invalid, not fully readable, or the first four bytes are not writable, then *IDA Stealth* will cause an exception. The *IDA* debugger will trap the exception, but the debugging session will be interrupted.

*IDA Stealth* hooks the debuggee's ntdll NtYieldExecution() function by replacing its first five bytes with a relative jump to an injected DLL. When the hook is reached, it calls the original ntdll NtYieldExecution() function, then returns successfully.

The plug-in hooks the debuggee's ntdll NtTerminateThread() function by replacing its first five bytes with a relative jump to an injected DLL. The hook detects an invalid handle by attempting to duplicate the handle. If the handle is valid, then the hook ignores the request and returns successfully. This could be considered a bug, because it disallows the intentional termination of user-created threads.

Similarly, *IDA Stealth* hooks the debuggee's ntdll NtTerminateProcess() function by replacing its first five bytes with a relative jump to an injected DLL. The hook detects an invalid handle by attempting to duplicate it. If the handle is valid, then the hook ignores the request and returns successfully. This could be considered a bug, because it disallows the intentional termination of user-created processes.

*IDA Stealth* hooks the debuggee's ntdll RtlGetVersion() function, if it exists (the API was introduced in *Windows 2000*), by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it checks if the RTL\_OSVERSIONINFOEXW format was requested. If the RTL\_OSVERSIONINFOEXW format was not requested, then the hook assumes that the requested format was the RTL\_OSVERSIONINFOW format. This behaviour is identical to that of *Windows XP*. However,

the different versions of *Windows* behave in different ways regarding this function. The exact problem is how to behave if the size field specifies a buffer that is not large enough to receive the full data. Specifically, *Windows 2000* always writes 0x14 bytes before checking the value in this field; *Windows XP* writes 0x14 bytes and the service pack string before checking the value in this field; *Windows Vista* and later versions write 0x14 bytes and the service pack string before checking the value in this field, but limit the copy to a maximum of 0xfe bytes. As a result, the function can behave in one of three ways.

Example code looks like this:

```
mov  eax, fs:[30h]
mov  d [eax+1f4h], offset 12
push offset 11
call  RtlGetVersion
...
11: db  16h dup (1)
12: db  100h dup (2), 0, 0
```

On *Windows 2000*, the byte at 11+0x14 has a value of 1, because the data is not copied at all. On *Windows XP*, the byte at 11+0x14 has a value of 2, because the data is copied irrespective of size. On *Windows Vista*, the byte at 11+0x14 has a value of 0, because the data is copied until the maximum length is reached, and then the value is zeroed because the string is too long. This behaviour allows *IDA Stealth* to be detected on specific platforms.

Example code looks like this:

```
call  GetVersion
cmp  al, 5
jb   not_supported
xchg ebx, eax
mov  eax, fs:[30h]
mov  d [eax+1f4h], offset 18
push offset 15
call  GetModuleHandleA
push offset 16
push  eax
call  GetProcAddress
push offset 17
call  eax
movzx ecx, b [offset 17+14h]
jecz 13
loop 12
;looks like Windows 2000
;fail if doesn't behave like it
cmp  bx, 5
11: je  14
;fail if unrecognised value
12: loop being_debugged
;looks like XP
;fail if doesn't behave like it
```

```

    cmp    bx, 105h
    jmp    l1
    ;looks like Windows Vista+
    ;fail if doesn't behave like it
13: cmp    bl, 6
    jb    being_debugged
14: ...
15: db    "ntdll", 0
16: db    "RtlGetVersion", 0
17: db    16h dup (1)
18: db    100h dup (2), 0, 0

```

*IDA Stealth* hooks the debugger's kernel32

DebugActiveProcess() function by replacing its first five bytes with a relative jump to the plug-in. When the hook is reached, it overwrites the entire contents of the debuggee's ntdll.dll code section with that of the debugger's ntdll.dll code section, whose size is specified by the SizeOfCode field in the PE header. This has the effect of removing any changes that the debuggee might have made, in an attempt to prevent a debugger from attaching to the process. However, this technique is detected very easily.

Example code looks like this:

```

    push  offset 13
    call  GetModuleHandleA
    push  offset 14
    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
    push  eax
    push  esp
    xor   eax, eax
    push  eax
    push  ebx
    push  offset 11
    push  eax
    push  eax
    call  CreateThread
    ...
11: pop  eax
    pop  eax
12: cmp  b [eax], 0c3h
    je   l2
    jmp  being_debugged
13: db  "ntdll", 0
    ;use a less common API
14: db  "DbgUserBreakPoint", 0

```

The plug-in installs a driver that makes the RDTSC instruction illegal when called from ring 3. The driver's name is 'rdtsceму' by default, but it can optionally be a random string value returned by either the kernel32 QueryPerformanceCounter() function or the kernel32 GetTickCount() function.

The driver intercepts the exception that occurs when the instruction is issued. When the exception occurs, the driver executes the RDTSC instruction in ring 0, and then returns a value that is controlled by the driver, as the elapsed time since the last time the RDTSC instruction was executed. The value has a delta applied to it, which is specified as part of a DeviceIoControl() call.

The author of *IDA Stealth* responded to the report. Some things were changed in version 1.1.1, such as adding support for the heap flags location for *Windows Vista*. However, the NtClose() problem remains on 64-bit *Windows*.

The author of *IDA Stealth* also released version 1.2 shortly afterwards, which introduced a new 'stealth' driver, but which also introduced some new bugs. The driver hooks the ntoskrnl NtQueryInformationProcess() function directly in the Service Descriptor Table. The hook calls the original ntoskrnl NtQueryInformationProcess() function, then checks if an error occurred. If no error occurred, then the hook checks if the ProcessInformationClass is the ProcessDebugPort class, and zeroes the port if so. There is a bug in that code, which is that the process handle is not checked. The correct behaviour would have been to zero the port only if the current process is specified.

The hook checks if the ProcessInformationClass is the ProcessDebugObjectHandle class. If it is, then the hook returns a handle value of one. There are three bugs in this code. The first is that the return value is incorrect. When the *IDA* debugger is active, the function will return successfully. That result alone is enough to reveal the presence of *IDA Stealth*. The second bug is that by changing the handle value, any legitimate use of that handle becomes impossible. The third bug is that the process handle is not checked. The correct behaviour would have been to return a failure with the correct error code, but only if the current process is specified.

The hook checks if the ProcessInformationClass is the ProcessDebugFlags class, and sets the flags to true if so, signifying that no debugger is present. There is a bug in this code, which is that the process handle is not checked. The correct behaviour would have been to zero the port only if the current process is specified.

Example code demonstrating these techniques was published in a previous paper [3].

The driver hooks the ntoskrnl NtSetInformationThread() function directly in the Service Descriptor Table. The hook intercepts attempts to call the ntdll NtSetInformationThread() function with the HideThreadFromDebugger class. The hook detects an invalid handle by attempting to duplicate the handle. If the handle is valid, then the hook ignores the request and returns successfully.

The author of *IDA Stealth* responded very quickly to the report. The bugs will be fixed in a future version.

The final part of this series will look at anti-unpacking by emulating.

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

## REFERENCES

- [1] <http://pferrie.tripod.com/papers/unpackers.pdf>.
- [2] <http://www.virusbtn.com/pdf/magazine/2008/200812.pdf>.
- [3] <http://www.virusbtn.com/pdf/magazine/2009/200901.pdf>.
- [4] <http://www.virusbtn.com/pdf/magazine/2009/200902.pdf>.
- [5] <http://www.virusbtn.com/pdf/magazine/2009/200903.pdf>.
- [6] <http://www.virusbtn.com/pdf/magazine/2009/200904.pdf>.
- [7] <http://www.virusbtn.com/pdf/magazine/2009/200905.pdf>.
- [8] <http://www.virusbtn.com/pdf/magazine/2009/200906.pdf>.
- [9] <http://www.virusbtn.com/pdf/magazine/2010/201005.pdf>.
- [10] <http://www.virusbtn.com/pdf/magazine/2010/201006.pdf>.
- [11] <http://www.virusbtn.com/pdf/magazine/2010/201007.pdf>.
- [12] <http://www.virusbtn.com/pdf/magazine/2010/201008.pdf>.
- [13] <http://www.virusbtn.com/pdf/magazine/2010/201009.pdf>.
- [14] <http://vx.netlux.org/lib/vrg02.html>.