

TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART ELEVEN

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 [1–11]. Now, the series continues with a look at tricks that are specific to debuggers and emulators.

In this article we look at some more *OllyDbg* plug-ins.

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

1. OLLYDBG PLUG-INS

OllyDbg supports plug-ins. A number of packers have been written to detect *OllyDbg*, so plug-ins have been written to attempt to hide it from those packers. The following is a description of some of those plug-ins, along with the vulnerabilities that could be used to detect them.

1.1 Stealth64

The *Stealth64* plug-in was described in [7]. What follows are the changes from the previous version, and a description of behaviour that is specific to more recent versions of *Windows*.

Stealth64 sets to zero the debuggee's PEB->BeingDebugged, in both the 32-bit and 64-bit PEBs. *Stealth64* sets the debuggee's PEB->NtGlobalFlag flags to zero, but only in the 32-bit PEB. The location of the NtGlobalFlag is different on the 64-bit version of *Windows Vista*. Since the 32-bit version is altered but the 64-bit version is not, the difference can be used to detect the presence of *Stealth64*.

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
;NtGlobalFlag
mov ecx, [eax+68h]
;64-bit PEB follows 32-bit PEB
cmp [eax+10bch], ecx
jne being_debugged
```

Stealth64 hooks the code in *OllyDbg* that is reached when a breakpoint exception occurs. It attempts to read the two bytes that exist three bytes before the current EIP value, and then checks for the 'CD2D' opcode ('INT 2D' instruction). If the opcode is seen, then *Stealth64* passes the exception to the debuggee instead of allowing *OllyDbg* to intercept it.

Stealth64 changes to read/write the page attributes of the KUSER_SHARED_DATA. This change is allowed

on 64-bit versions of *Windows*, and it has the effect of decoupling the page from its kernel-mode counterpart. The result is that the data in the page is no longer updated. This change affects APIs such as the kernel32 GetTickCount() function, which read their values from this page. The change causes the API to always return the same value to that process (other processes are not affected).

Stealth64 changes the address in each of the debuggee's thread's TEB->Wow32Reserved field values, to point to a dynamically allocated block of memory. That field is undocumented, but it normally points into a function within the wow64cpu.dll, which orders the parameters for a 64-bit system call, and then falls into the wow64cpu TurboDispatchJumpAddressStart() function to perform the transition to kernel mode. By changing this field value, *Stealth64* creates a clean single point of interception for all system calls.

The block that *Stealth64* allocates contains code to watch for particular system table indexes. *Stealth64* knows the appropriate index values for *Windows XP*, *Windows Server 2003*, *Windows Server 2008*, *Windows Vista* and *Windows 7*. The indexes that are intercepted are: NtQueryInformationProcess, NtQuerySystemInformation, NtSetInformationThread, NtClose, NtOpenProcess, NtQueryObject, FindWindow, BlockInput, BuildHwndList, NtProtectVirtualMemory, NtQueryInformationProcess (again), GetForegroundWindow and GetWindowThreadProcessId. The duplication of the NtQueryInformationProcess index is a bug. It was intended to be the NtQueryVirtualMemory index.

If the NtQueryInformationProcess index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then exits if an error occurs, or if the handle does not refer to the debuggee. Otherwise, the hook checks the ProcessInformationClass parameter. If the ProcessDebugObjectHandle class is specified, then the hook zeroes the handle and then tries to return STATUS_PORT_NOT_SET (0xC0000353). However, there is a bug in this code, which results in the status always being STATUS_SUCCESS.

If the NtQuerySystemInformation index is seen, the ReturnLength is zero, the SystemInformationClass is the SystemProcessInformation class, and this is the first time that the index has been seen, then the block allocates some data and uses a Thread Local Storage slot to hold it. This is the correct method to hold private thread-specific data.

If the NtSetInformationThread index is seen, and the ThreadInformationClass is the HideThreadFromDebugger class, then the hook changes the class to the ThreadSwitchLegacyState class before calling the original TEB->Wow32Reserved pointer. This avoids the need to

check the handle. The ThreadSwitchLegacyState class handler checks the handle, and if the handle is valid, then it simply sets a flag in the object before returning.

If the NtClose index is seen, then the hook calls the ntoskrnl NtQueryObject() function to verify that the handle is valid. If it is valid, then the hook calls the ntoskrnl NtClose() function. Otherwise, it returns STATUS_INVALID_HANDLE (0xC0000008). However, disabling the exception in this way, 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 *Stealth64*.

If the NtOpenProcess index is seen, then the hook checks that the ClientId parameter points to a valid memory location that is both readable and writable. If the memory location is valid, and if the request is for the *OllyDbg* process ID, then the hook changes the process ID to the parent of *OllyDbg*, before calling the original TEB->Wow32Reserved pointer.

If the NtQueryObject index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and exits if an error occurs. Otherwise, the hook checks the ObjectInformationClass parameter. If the ObjectInformationClass is the ObjectAllTypesInformation class, then the hook searches the returned buffer for all objects whose length is 0x16 bytes and whose name is 'DebugObject'. If this is found, then the hook zeroes the object and handle counts.

If the BlockInput index is seen, then the hook checks if the new state is the same as the old state. If they are different, then the hook saves the new state and returns success. Otherwise, the hook returns a failure. This is the correct behaviour, since *Windows* behaves in an identical manner. It will not allow the input to be blocked twice, nor will it allow the input to be enabled twice.

If the NtProtectVirtualMemory index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then exits if an error occurs, or if the handle does not refer to the debuggee. Otherwise, the hook protects as read-write (that is, non-executable) any pages that were marked for DEP-breakpoint support. This could be considered a bug, since the original page protection might have been something other than read-write, which could lead to some unexpected behaviour.

If the NtQueryVirtualMemory index could be seen, then the hook would have checked the requested page before it called the original TEB->Wow32Reserved pointer, and then exited if an error occurred. There are two bugs in this code. The first is that the hook checks if the requested page exactly matches the first page that was marked for DEP-breakpoint support. The problem is that a region spanning multiple pages can be marked for DEP-breakpoint support. As

a result, requesting information about the second or subsequent pages would result in the altered page protection being returned. The second bug is that the status is always set to STATUS_SUCCESS, even if an error occurs.

There is a further problem with that hook, which is that the pages that are marked for DEP-breakpoint support are assumed to retain their original protection forever. There is a single variable that holds the protection for the entire range. However, if an individual page is set to a different protection value, this change will not be visible, and the original protection will always be returned when requested.

If the GetForegroundWindow index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then exits if an error occurs. Otherwise, the hook checks if the foreground window belongs to *OllyDbg*. If it does, then the hook returns the desktop window instead.

If the GetWindowThreadProcessId index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then checks if the returned value matches the process ID of *OllyDbg*. If it does, then the hook returns the process ID of the parent process instead. However, there is a bug in this code, which is that the hook does not check which kind of information was requested. The same function can request either a process ID or a thread ID. Since these IDs are not unique across the system, it is possible to have a thread ID within one process that will match the process ID of *OllyDbg*. The result would be that the hook would return a possibly invalid ID, with unpredictable results. There is a second problem associated with this behaviour, which is that the hook does not protect against the main thread ID of *OllyDbg* being found. Since the thread ID can be found, the corresponding thread handle can be retrieved, and then the thread can be opened. Once the thread has been opened, it can be suspended, for example, which will cause the debugging session to halt.

The author of *Stealth64* is investigating the report.

1.2 Poison

Poison patches the debuggee's user32 BlockInput() function to simply return. This behaviour is a bug, since the return code is never set.

Poison sets to zero the PEB->BeingDebugged and the low byte of the PEB->NtGlobalFlag flags. It patches the debuggee's kernel32 CheckRemoteDebuggerPresent() function to always return zero.

It patches the debuggee's user32 FindWindowA() and FindWindowExA() functions to always return zero, irrespective of parameters that were passed to the functions. The user32 FindWindowW() and FindWindowExW() functions are not patched.

Poison patches the debuggee's kernel32 OutputDebugStringA() to always return success.

Poison sets the PEB->Heap->ForceFlags flags to zero, and sets the PEB->Heap->Flags flags to HEAP_GROWABLE.

Poison retrieves the heap information from PEB->NumberOfHeaps and PEB->ProcessHeaps. It applies the Heap->ForceFlags and Heap->Flags patch to each heap. It also zeroes the last four bytes of the first page of each heap. This last change is a serious bug, since the first page contains information about the heap itself. An off-by-one bug also exists, resulting in an attempt to apply the patch to a memory location that does not point to a heap. The value in that memory location is usually zero, but it can be changed by the debuggee, or the debuggee can allocate memory at virtual address zero. In either case, *Poison* would apply the patch to wherever the pointer points. This is possible because the plug-in does not run immediately. Instead, it requires user interaction in order to start, which means that the debuggee might have been executing for some time before the plug-in is executed manually.

Poison hooks the debuggee's ntdll NtSetInformationThread() function by replacing its first five bytes with a relative jump to a dynamically allocated block of memory. That block intercepts attempts to call the ntdll NtSetInformationThread() function with the HideThreadFromDebugger class, and then simply returns. This behaviour is a bug, since the return code is never set. There is another bug in this code, which is that if any other class is seen, then the hook calls the original handler, but using a hard-coded index value of 0xe5. This corresponds to the NtSetInformationThread index for Windows XP SP3 only. On other platforms, the resulting behaviour is unpredictable.

Poison searches infinitely within the debuggee's ntdll NtQueryInformationProcess() function code for the 'FF12' opcode ('CALL [EDX]' instruction), and then replaces it with an 'E9' opcode ('JMP' instruction), to point to a dynamically allocated block of memory. *Poison* assumes that the first match is the correct one, even though it might be a constant portion of another instruction. The block intercepts attempts to call the ntdll NtQueryInformationProcess() function with the ProcessDebugPort class, and tries to return zero for the port in that case. The block also checks if the ntdll NtQueryInformationProcess() function was called with the ProcessBasicInformation class, and tries to replace the InheritedFromUniqueProcessId with the process ID of Explorer.exe. However, there is a bug in both codes, which is that there is no check that the ProcessInformation parameter points to a valid memory address, or that the entire ProcessInformationLength range is writable. If either the ProcessInformation pointer or the ProcessInformationLength is invalid for some reason,

then *Poison* will cause an exception. *OllyDbg* will trap the exception, but the debugging session will be interrupted.

The correct behaviour would have been to zero the port, or perform the replacement, only if the function returned successfully, and only if the current process is specified. However, the current process can be specified in ways other than the pseudo-handle that is returned by the kernel32 GetCurrentProcess() function, and that must be taken into account. There is another bug in the code, which is that if any other class is specified, then the block simply returns without setting a return value, and does not call the original handler.

Poison patches the debuggee's ntdll DebugBreak() function to simply return. This behaviour is a bug because no exception will be raised if the function is called.

Poison hooks the debuggee's ntdll NtOpenProcess() function by replacing its first five bytes with a relative jump to a dynamically allocated block of memory. That block intercepts attempts to call the ntdll NtOpenProcess() function with the process ID of the debuggee, and then changes the process ID to zero before calling the original handler. The check for the process ID of the debuggee is probably a bug. The more sensible process ID for which to deny access would be that of *OllyDbg*. However, there is a definite bug in this code, which is that when the hook calls the original handler, it uses a hard-coded index value of 0x7a. This corresponds to the NtOpenProcess index for Windows XP SP3 only. On other platforms, the resulting behaviour is unpredictable.

Poison patches the debuggee's kernel32 CreateThread() to simply return. This will obviously break any process which wants to create a thread.

Poison patches the debuggee's kernel32 GetTickCount() function in one of two ways. The first method zero-extends the low byte of the KUSER_SHARED_DATA->TickCountLowDeprecated field value into the returned dword and the undocumented extended dword. It then adds 0x431 to the returned dword. The result is a tick count that always has the form 00000abb, and where a is either 4 or 5. Thus, time appears to move very slowly. The second method simply returns a constant tick count of 0x50400.

Poison hooks the debuggee's ntdll KiRaiseUserExceptionDispatcher() function by replacing its first five bytes with a relative jump to a dynamically allocated block of memory. That block intercepts attempts to call the ntdll KiRaiseUserExceptionDispatcher() function with the EXCEPTION_INVALID_HANDLE (0xC0000008) value, and returns zero if so. However, there is a bug in this code, which is that if another exception value is seen, then the hook creates a stack frame and then uses a hard-coded branch directly into the middle of the original function. This obviously assumes that the stack frame size is correct, and that the layout of the function will never

change. Otherwise, the destination of the branch instruction might be the middle of an instruction.

Poison patches the debuggee's ntdll NtYieldExecution() function to always return a status. This hides *OllyDbg* from the NtYieldExecution() detection method.

Poison patches the debuggee's kernel32 Process32NextW() function in one of two ways. The first method searches infinitely within the debuggee's kernel32 Process32NextW() function code for the 'C2080090' opcode ('RET 8' and 'NOP' instructions), and then replaces it with an 'E9' opcode ('JMP' instruction), to point to a dynamically allocated block of memory. *Poison* assumes that the first match is the correct one, even though it might be a constant portion of another instruction. It also checks only those four bytes, but overwrites them with a jump instruction that is five bytes long. The block examines the returned buffer for the process ID of either the debuggee or *OllyDbg*. If the process ID of the debuggee is seen, then the block replaces it with the process ID of Explorer.exe. If the process ID of *OllyDbg* is seen, then the block replaces it with zero. This has the effect of making the debuggee invisible to itself, which is a strange thing to do. The second method patches the debuggee's kernel32 Process32NextW() function to always return zero.

Poison patches the debuggee's kernel32 Module32NextW() and kernel32 EnumWindows() functions to always return zero.

Poison searches infinitely within the debuggee's ntdll NtQueryObject() function code for the 'FF12' opcode ('CALL [EDX]' instruction), and then replaces it with an 'E9' opcode ('JMP' instruction), to point to a dynamically allocated block of memory. *Poison* assumes that the first match is the correct one, even though it might be a constant portion of another instruction. The block intercepts attempts to call the ntdll NtQueryObject() function with the ObjectAllTypesInformation class, and tries to zero out the entire returned data. However, there is a bug in that code, which is that there is no check that the ObjectInformation parameter points to a valid memory address, or that the entire ObjectInformationLength range is writable. If either the ObjectInformation pointer or the ObjectInformationLength is invalid for some reason, then *Poison* will cause an exception. *OllyDbg* will trap the exception, but the debugging session will be interrupted. It would have been better to zero out the entire returned data only if the function returned successfully. The correct behaviour would be to parse the returned data to find the DebugObject, if it exists, and then zero out the individual handle counts, but only if the function returned successfully. This arbitrary erasure is an obvious sign that *Poison* is active.

Poison patches the debuggee's kernel32 QueryPerformanceFrequency(), ntdll

NtQueryPerformanceFrequency(), kernel32 QueryPerformanceCounter() and ntdll NtQueryPerformanceCounter() functions to always return zero.

Poison patches a breakpoint handler in *OllyDbg* to always write a zero to the debuggee's PEB->BeingDebugged field.

Poison patches the FPU handler in *OllyDbg* from the 'DF38' opcode ('FISTP QWORD PTR [EAX]' instruction) to the 'DB38' opcode ('FSTP TBYTE PTR [EAX]' instruction). This causes the disassembly to be incorrect, including for values which would not have triggered the problem.

Poison patches some code in *OllyDbg* to zero out the start-up information that is used to load the debuggee. The effect is essentially to zero the flags and ShowWindow parameters, which could have been done in a far more elegant manner.

Poison changes the window caption in *OllyDbg* from 'OllyDbg' to 'POISON'. The name of the debuggee is removed, and the 'CPU' window caption is changed to '[Professional Edition]'.

The next part of this series will look at anti-unpacking tricks that are specific to a range of other debuggers including *HideToolz*, *Obsidian* and *Turbo Debug32*.

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>.