TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART SIX

Peter Ferrie
Microsoft, USA

New anti-unpacking tricks continue to be developed as the older ones are constantly being defeated. This series of articles (see also [1–5]) describes some tricks that might become common in the future, along with some countermeasures.

This article concentrates on anti-debugging tricks that target plug-ins for the *OllyDbg* debugger. All of the techniques described here were discovered and developed by the author.

OllyDbg plug-ins

OllyDbg is perhaps the most popular user-mode debugger. A number of packers have been written that are able to detect *OllyDbg*, so plug-ins have been created to attempt to hide it from those packers.

Last month we looked at *antiAnti*, *HideDebugger*, *HideOD*, *IsDebugPresent*, *Olly Advanced* and *OllyICE*. In this article we look at some more *OllyDbg* plug-ins and the vulnerabilities that could be used to detect them.

Olly Invisible

Olly Invisible hooks the code in *OllyDbg* that is reached when it is formatting the kernel32 OutputDebugStringA() string, and then attempts to replace all '%' characters with ' ' in the message. However, a bug in the routine causes it to miss the last character in the string.

The plug-in hooks the debuggee's kernel32 OutputDebugStringA() function by replacing the first six bytes with an indirect jump to a dynamically allocated block of memory. This block attempts to replace all '%' characters with '_' in the message.

Similarly, *Olly Invisible* hooks the debuggee's kernel32 OutputDebugStringW() function by replacing the first six bytes with an indirect jump to a dynamically allocated block of memory. This block attempts to replace all '%' characters with '_' in the message.

The plug-in hooks the debuggee's kernel32 IsDebuggerPresent() function in the same way – by replacing the first six bytes of the function with an indirect jump to a dynamically allocated block of memory. In this case the block always returns zero, regardless of the value in the PEB->BeingDebugged flag.

Olly Invisible hooks the debuggee's ntdll NtQueryInformationProcess() function in the same way again, replacing the first six bytes of the function with an indirect jump to a dynamically allocated block of memory. This block calls the original ntdll NtQueryInformationProcess() function, and then checks whether an error occurred. If no error occurred, then the block checks if the ProcessInformationClass is the ProcessDebugPort class, and that the ProcessInformation parameter is non-zero, and then checks that four bytes are writable at the specified memory address. If all of these requirements are met, *Olly Invisible* writes a zero to the memory address at which the ProcessInformation parameter points. This method is almost unflawed, but it omits a check of whether 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.

If possible, *Olly Invisible* patches the debuggee's ntdll CsrGetProcessId() function, so that it always returns zero. However, since this function should never return zero, such a result is a sure sign that the plug-in is present.

Olly Invisible hooks the debuggee's ntdll NtQuerySystemInformation() function by replacing the first six bytes with an indirect jump to a dynamically allocated block of memory. This block calls the original ntdll NtQuerySystemInformation() function, and then checks if an error occurred. If no error occurred, it checks if the SystemInformationClass is the SystemProcessInformation class. If it is, then the block searches within the returned process list for processes with the image name 'OllyDbg.exe'. If any are found, then the block adjusts the list so that it skips those entries. However, the entries themselves are untouched, and can be found by a brute-force search of the returned buffer.

Olly Invisible hooks the debuggee's ntdll NtReadVirtualMemory() function by replacing the first six bytes of the function with an indirect jump to a dynamically allocated block of memory. This block calls the original ntdll NtReadVirtualMemory() function, and then checks if an error occurred. If no error occurred, it checks if the read includes the address of a hooked function. If it does, then the block restores the original bytes of the function in the returned buffer, thus achieving in-memory stealth for remote processes. However, there are three problems in the code.

The first problem is in the bounds check: *Olly Invisible* only checks if the read includes the address of the first byte of a hooked function. This means that if the read begins one byte after the start of the hooked function, then the hook will be visible. The second problem is that *Olly Invisible* does not check how many bytes have been read, but always attempts to restore the six altered bytes. Thus, even if only one byte was read, six bytes will be written to the buffer. If the buffer is at the end of a page or in a sensitive location, then an exception or memory corruption could occur as a result. The third problem is that *Olly Invisible* does not check the process handle for which the request was made, which can lead to the 'stealthing' of the memory of a completely different process. The correct behaviour would be to restore the bytes 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.

Olly Invisible sets the debuggee's PEB->BeingDebugged flag to zero.

The author of *Olly Invisible* has not responded to the report.

PhantOm

The *PhantOm* plug-in changes the 'OllyDbg - <filename> - [CPU]' string in *OllyDbg* to 'PhantOm - [CPU]'.

It changes the 'CPU -' string either to the one specified in the phantom.ini file, or to 'o_O' if no string is specified. It changes the '%smodule' string to '%sm0dule', and changes the 'NULL thread' string to 'NULL thr3ad'.

PhantOm changes the export address for the debuggee's ntdll NtQueryInformationProcess() function so that it points into the file header of ntdll.dll. It also changes the corresponding import address in the debuggee's kernel32.dll to point into the file header of ntdll.dll. It copies the original ntdll NtQueryInformationProcess() function code into the file header of ntdll.dll, and then appends some code to the copied function. The appended code checks the ProcessInformationClass parameter. If the ProcessTimes class is specified, then the hook returns an error. The purpose of the change to kernel32.dll is to hook the kernel32 GetProcessTimes() function implicitly. The ProcessTimes class can be used to expose the length of time that a user requires to debug an application, so by hiding this information, it hides *OllyDbg* too.

PhantOm aims to patch the debuggee's user32 BlockInput() function code to always return successfully, but this code does not work in *Windows 2000* and earlier because of an apparently reversed conditional statement.

PhantOm erases the dwX, dwY, dwXSize, dwYSize, dwXCountChars, dwYCountChars and dwFillAttribute fields from the RTL_USER_PROCESS_PARAMETERS block. These characteristics are checked by the ChupaChu debugger test, which also checks whether bit 7 is set in the dwFlags field. However, due to a bug, the latter check always fails. If it were not for the bug, the ChupaChu test would detect the plug-in.

PhantOm attempts to hook the debuggee's ntdll KiUserExceptionDispatcher() function by replacing an 0xE8 opcode ('CALL' instruction) with an 0xE9 opcode ('JMP' instruction) at a fixed location within the routine. This behaviour is a bug, because in *Windows Vista* the routine has an additional instruction prepended to it, meaning that the required instruction is in a different location. However, if the hook is successful, then when an exception occurs, the hook saves the state of the debug registers into a private memory region. The hook then swaps in the previous debug register values before passing the exception to the debuggee. This tricks the debuggee into thinking that any changes it makes are current. *PhantOm* also attempts to hook the ntdll NtContinue() function in order to save the updated debug register values on return from the debuggee. However, a bug exists in the hooking code. The hook checks for the correct instruction before replacing it, but due to an incorrect conditional assignment, it performs the replacement regardless of the result.

PhantOm hooks the debuggee's kernel32 GetTickCount() function by replacing the first five bytes of the function with a relative jump to a dynamically allocated block of memory. This block intercepts attempts to call the kernel32 GetTickCount() function, and then returns a tick count that is incremented by one each time it is called, regardless of how much time has passed.

PhantOm patches __fuistq() in *OllyDbg* to avoid the floating-point operations error. It does this by skipping the data conversion. This is not the proper way to avoid the problem, however, since no values are converted as a result. A better fix would be to change the floating-point exception mask to ignore such errors. This can be achieved by changing the dword at file offset 0xCB338 from 0x1332 to 0x1333, or just by loading that value manually into the control word of the FPU.

PhantOm patches the code in *OllyDbg* that is reached when it is formatting the kernel32 OutputDebugStringA() string. The patch prevents the debugger from formatting the message.

PhantOm hooks the code in *OllyDbg* that is reached when a debug event occurs. When the hook is reached, it checks for the following events:

- If the DBG_PRINTEXCEPTION_C (0x40010006) exception is seen, then the hook returns a status that the exception was not handled. This hides *OllyDbg* from the kernel32 GetLastError() detection method.
- If the EXCEPTION_ACCESS_VIOLATION (0xC0000005) or EXCEPTION_GUARD_PAGE (0x80000001) exception is seen and is not within the bounds of a memory breakpoint, then the hook returns a status that the event was not handled. This hides *OllyDbg* from the guard page detection method.

- If the EXCEPTION_ ILLEGAL_INSTRUCTION (0xC000001D), EXCEPTION_INVALID_LOCK_SEQUENCE (0xC000001E) or EXCEPTION_INTEGER_DIVIDE_BY_ZERO (0xC00000094) exception is seen, then *PhantOm* returns a status that the event was not handled. This prevents *OllyDbg* from breaking on several common conditions.

PhantOm installs a driver which hooks the NtQueryInformationProcess(), NtOpenProcess(), NtClose(), NtSetInformationThread(), NtYieldExecution(), NtQueryObject(), NtQuerySystemInformation() and NtSetContextThread() functions in ntoskrnl.exe by name, and the GetWindowThreadProcessId(), EnumWindows(), FindWindowA() and GetForegroundWindow() functions in ntoskrnl.exe by service table index. What happens next depends on the hook that is called:

- When the NtQueryInformationProcess() function is called, the hook checks the ProcessInformationClass parameter. If the ProcessDebugPort class was specified, then the hook zeroes the debug port, but without checking the process handle. The correct behaviour would be to zero the port 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.

If the ProcessBasicInformation class was specified, then the hook replaces the process ID of *OllyDbg* 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 *OllyDbg*'s parent.

- When the NtOpenProcess() function is called, the hook checks if the process ID to open matches the process ID of *OllyDbg* or CSRSS.EXE, and returns an error in the latter case.
- When the NtClose() function is called, the hook checks for a valid handle before attempting the close. This hides *OllyDbg* from the CloseHandle(invalid) detection method.
- When the NtSetInformationThread() function is called, the hook checks if the HideThreadFromDebugger class has been specified, and returns success if that is the case. There is a bug in this code, which is that if an invalid handle is passed to the function, then an error code should be returned. A successful return would be an indication that *PhantOm* is running.
- When the NtYieldExecution() function is called, the hook always returns a status. This hides *OllyDbg* from the NtYieldExecution() detection method.
- When the NtQueryObject() function is called, the hook checks for the ObjectAllTypesInformation class, and then erases all the returned information if it is specified.
- When the NtQuerySystemInformation() function is called, the hook checks the SystemInformationClass parameter. If the SystemKernelDebuggerInformation class is specified, then the hook erases all of the returned information. If the SystemProcessInformation class is specified, then the hook adjusts the list to skip those entries. However, the entries are untouched and can be found by a brute-force search of the returned buffer.
- When the NtSetContextThread() function is called, the hook clears the CONTEXT_DEBUG_REGISTERS flag from the ContextFlags field before completing the call. This prevents the debug register values from being returned, and hides *OllyDbg* from the debug registers detection method.
- When the GetWindowThreadProcessId() function is called, the hook checks whether the process ID matches the process ID of *OllyDbg*, and returns zero if that is the case. This technique hides *OllyDbg* from the window handle detection method.
- When the EnumWindows() function is called, the hook removes from the list all windows whose process ID matches that of *OllyDbg*. This technique hides *OllyDbg* from the window handle detection method.
- When the FindWindow() function is called, the hook checks whether the returned window handle belongs to *OllyDbg*, and returns zero if that is the case.
- When the GetForegroundWindow() function is called, the hook checks whether the returned window handle belongs to *OllyDbg*, and returns the previous foreground window handle in that case.

PhantOm installs a driver that makes the RDTSC instruction illegal when called from ring 3. 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 uses the low byte of the returned value as the time elapsed since the last time the RDTSC instruction was executed. This has the effect of slowing perceived time, and hides *OllyDbg* from the RDTSC detection method.

PhantOm sets the debuggee's PEB->BeingDebugged flag to zero.

One of the authors of *PhantOm* responded to the report: the BlockInput() bug will be fixed in a future version; the KiUserExceptionDispatcher() and NtContinue() bugs will remain, because *Windows Vista* is not supported.

Stealth64

The *Stealth64* plug-in forces *OllyDbg* to ignore the OptionalHeader bug described in [6].

Stealth64 patches the code in *OllyDbg* that is reached when it reads the debuggee's imported function names. The patch stops *OllyDbg* from displaying an error message if an imported function name cannot be read.

The plug-in patches the code in *OllyDbg* that is reached when it parses the debuggee's Import Table. The patch stops *OllyDbg* from displaying an error message if the import table appears to be corrupted.

Stealth64 patches the code in *OllyDbg* that is reached when it parses the debuggee's Base Relocation Table. The patch stops *OllyDbg* from applying relocations. However, this also prevents *OllyDbg* from debugging certain files.

Stealth64 handles the exception-priority trick described in [1] by forcing a single-step exception to occur in the ntdll KiUserExceptionDispatcher() function.

The plug-in sets the debuggee's PEB->BeingDebugged and PEB->NtGlobalFlag flags to zero.

Stealth64 hooks the debugger's kernel32 CreateProcessA() function. The hook defines and sets the '_NO_DEBUG_HEAP' environment variable to one, before calling directly into the kernel32 CreateProcessInternalA() function. This environment variable forces a process to use a standard heap instead of a debugging heap, even if the process is being debugged.

Stealth64 removes the SeDebugPrivilege from the process token.

Stealth64 hooks the debuggee's ntdll KiUserExceptionDispatcher() function. When an exception occurs, the hook saves the state of the debug registers into a private memory region if the 'ProtectDRX' option is enabled. The hook swaps in the previous debug register values if the 'HideDRX' option is enabled, before passing the exception to the debuggee. This tricks the debuggee into thinking that any changes it makes are current. *Stealth64* also hooks the ntdll NtContinue() function, in order either to save the updated debug register values on return from the debuggee if the 'HideDRX' option is enabled, or to swap back the original debug register values if the 'ProtectDRX' option is enabled.

Stealth64 searches within up to 256 bytes of the debugger's ntdll DbgUiConvertStateChangeStructure() function for a reference to the DBG_PRINTEXCEPTION_C (0x40010006) exception, followed by an 0x75 opcode ('JNE' instruction). If the sequence is found, then it replaces the 0x75 opcode with an 0xEB opcode ('JMP' instruction). The ntdll DbgUiConvertStateChangeStructure() function was introduced in *Windows XP*, but *Stealth64* runs only in *Windows Vista64*, so there is no problem with earlier versions of *Windows*. The effect of the patch is to prevent the OUTPUT_DEBUG_STRING_EVENT debug event from being delivered to the debugger. Instead, a generic EXCEPTION_DEBUG_EVENT debug event is delivered to the debugger. This hides *OllyDbg* from the GetLastError() detection method. However, there is a bug in the search routine, which assumes that all five bytes can be read. If the read accesses out-of-bounds memory, then *OllyDbg* will crash.

Stealth64 intercepts the EXCEPTION_GUARD_PAGE (0x80000001) exception and checks the address at which the fault occurred. If the fault is not within the bounds of a memory breakpoint, then the hook returns a status that the event was not handled. This hides *OllyDbg* from the guard page detection method.

Stealth64 changes the address in each of the debuggee 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. This act ties *Stealth64* to a specific version of *Windows Vista64*. The indexes that are intercepted are: NtQueryInformationProcess, NtQuerySystemInformation, NtSetInformationThread, NtClose, NtOpenProcess, NtQueryObject, FindWindow, BlockInput, NtQueryPerformanceCounter, BuildHwndList, NtProtectVirtualMemory and NtQueryVirtualMemory. If none of these indexes is seen, and if the 'HandleSingleStepExceptions' option is enabled, then *Stealth64* will register a Vectored Exception Handler. That handler consumes EXCEPTION_SINGLE_STEP (0x80000004) exceptions that occur in the region of memory that includes the injected code.

If the NtQueryInformationProcess index is seen, then the hook calls the original TEB->Wow32Reserved pointer and checks if the function has succeeded. If it has, then the hook checks the ProcessInformationClass parameter. If the ProcessDebugPort class is specified, then the hook zeroes the port and returns success. If the ProcessDebugObjectHandle class is specified, then the hook zeroes the handle and returns STATUS_PORT_NOT_SET (0xC0000353). If the ProcessDebugFlags class is specified, then the hook sets the flags to true, signifying that no debugger is present, and returns success. The correct behaviour for these three classes is for the changes to be applied 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.

If the ProcessBasicInformation class is specified, then the hook replaces the process ID of *OllyDbg* 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 *OllyDbg*'s parent.

If the NtQuerySystemInformation index is seen, the ReturnLength is zero and the SystemInformationClass is the SystemProcessInformation class, then the hook uses the TIB->ArbitraryDataSlot field to hold the returned length. There is a bug here, which is that the previous value in that field is not saved, and the hook always zeroes it before returning. The problem with this approach is that it can be detected by malware that sets the TIB->ArbitraryDataSlot field value to non-zero, then calls the ntdll NtQuerySystemInformation() function with no ReturnLength parameter. *Stealth64* is revealed because the TIB->ArbitraryDataSlot field value is zero.

In any case, the hook calls the original TEB->Wow32Reserved pointer, and then checks if the function has succeeded. If it has, then the hook checks that the 'Fake Parent' option is enabled. If it is, then the hook replaces the process ID of *OllyDbg* with the process ID of EXPLORER.EXE in the InheritedFromUniqueProcessId field. This could be considered another bug, since the true parent might not be *Explorer* (as before, the proper behaviour would be to use the process ID of *OllyDbg*'s parent).

The hook also checks if the 'NtQuerySystemInformation' option is enabled. If it is, then the hook parses the returned process list. The hook deletes the entry that corresponds to *OllyDbg* by copying the entries that follow over the top and then reducing the returned length.

If the NtSetInformationThread index is seen, and the ThreadInformationClass is the HideThreadFromDebugger class, then the hook returns success. There is a bug in this code, which is that if an invalid handle is passed to the function, then an error code should be returned. A successful return would be an indication that *Stealth64* is running.

If the NtClose index is seen, then the hook calls the ntdll NtQueryObject() function to verify that the handle is valid. If it is, then the hook calls the ntdll NtClose() function. Otherwise, it returns STATUS_INVALID_HANDLE (0xC0000008).

If the NtOpenProcess index is seen, then the hook attempts to replace the process ID of *OllyDbg* with the process ID of EXPLORER.EXE in the address to which the ClientId parameter points. However, there are three bugs here: the first is that the hook does not check if the ClientId parameter points to a valid memory location. An invalid memory address causes an exception that can be intercepted by the debuggee. Such an exception is a sure sign that *Stealth64* is running. The second bug is that the hook does not check if the ClientId parameter points to a writable memory location prior to attempting to replace the process ID. Writing to a read-only memory address causes an exception that can be intercepted by the debuggee. Such an exception is an indication that *Stealth64* is running. The third bug is that the hook zeroes the upper 32 bits of the quadword to which the ClientId parameter points. This can allow the function to succeed in places where it should fail. A successful return in that case is another sign that *Stealth64* is running.

If the NtQueryObject index is seen, then the hook calls the original TEB->Wow32Reserved pointer, then checks if the function has succeeded. If it has, then the hook checks if the ObjectInformationClass is the ObjectAllTypesInformation class. If it is, then the hook searches the returned buffer for all objects whose length is 0x16 bytes, and then zeroes the object counts, without checking the object name. This is a bug, since there could be other objects with the same name length, and their handle counts will also be zeroed.

If the FindWindow index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then checks if the function has succeeded. If it has, then the hook calls the user32 GetWindowThreadProcessId() function for the returned window handle. The hook returns zero if the returned process ID matches the process ID of *OllyDbg*.

If the BlockInput index is seen, then the hook simply returns. This behaviour is a bug, since the return code is never set.

If the NtQueryPerformanceCounter index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then checks if the function has succeeded. If it has, then the hook returns a tick count that is incremented by one each time it is called, regardless of how much time has passed.

If the BuildHwndList index is seen, then the hook calls the original TEB->Wow32Reserved pointer, and then checks if the function has succeeded. If it has, then the hook parses the returned hwnd list and then deletes the entry that corresponds to *OllyDbg* by copying the entries that follow over the top, and then reducing the returned length.

If the NtProtectVirtualMemory index is seen, then the hook checks if the ProcessHandle parameter corresponds to the GetCurrentProcess() pseudo-handle. If it does, then the hook checks if the value in the memory location to which the BaseAddress parameter points matches the location of the internal breakpoint address that *Stealth64* uses. If it does, then the hook returns success. There are three bugs in this code, and one behaviour that could be considered a bug. The first bug is that the hook does not check if the BaseAddress parameter points to a valid memory location. An invalid memory address causes an exception that can be intercepted by the debuggee. Such an exception is a good indication that *Stealth64* is running. The second bug is that the BaseAddress parameter can span the region that is protected by the internal breakpoint, and as a result the comparison will fail. The third bug is that to return success if the comparison succeeds might be incorrect behaviour if the NewAccessProtection parameter specifies an invalid protection value. In that case, an error code should be returned instead. The behaviour that could be considered a bug is that the ProcessHandle parameter might contain a handle to the current process, as returned by the kernel32 OpenProcess() function. This handle will not be recognized as belonging to the current process.

If the NtQueryVirtualMemory index is seen, then the hook checks if the BaseAddress parameter matches the location of the internal breakpoint address that *Stealth64* uses, and that the VirtualMemoryInformationClass parameter is zero. If those checks succeed, then the hook calls the original TEB->Wow32Reserved pointer, and attempts to set the value in the VirtualMemoryInformation->Protect field to Executable/Readable/Writable, if the VirtualMemoryInformation parameter has been specified.

There are four bugs in this code. The first is that the hook does not check if the function call succeeded. The second bug is that the hook does not check if the VirtualMemoryInformation parameter points to a valid memory location. An invalid memory address causes an exception that the debuggee can intercept. Such an exception is a sure sign that *Stealth64* is running. The third bug is that the hook does not check if the VirtualMemoryInformation parameter points to a writable memory location prior to attempting to write the VirtualMemoryInformation->Protect field value. Writing to a read-only memory address causes an exception that the debuggee can intercept. Such an exception is a sure sign that *Stealth64* is running. The fourth bug is that the hook does not check the process handle for which the request was made, which can lead to lying about the memory state of a completely different process. The correct behaviour would have been to check 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.

A HandleInt2D option exists but it is not supported.

The author of *Stealth64* responded to the report, and the bugs will be fixed in a future version.

Olly's Shadow

Olly's Shadow is a patched and renamed version of *OllyDbg*. Since it is renamed, it hides *OllyDbg* from the standard FindWindow() and process enumeration detection techniques. *Olly's Shadow* does not export any functions, which avoids another common detection method on the export name table. However, this prevents the use of plug-ins, unless they use hard-coded addresses.

Olly's Shadow behaves like *Olly Invisible* with respect to the OutputDebugString handling, complete with the same bug: *Olly's Shadow* hooks the code in *OllyDbg* that is reached when *OllyDbg* is formatting the kernel32 OutputDebugStringA() string, and then attempts to replace all '%' characters with ' ' in the message. However, a bug in the routine causes it to miss the last character in the string.

Olly's Shadow changes the options that are used when loading symbols, and then disables the name merging. This avoids several problems with corrupted symbol files, including the dbghelp.dll bug described in [1].

Olly's Shadow changes the class name from 'OLLYDBG' to 'SHADOW', and the window title from 'OllyDbg' to 'Shadow'.

The author of *Olly's Shadow* could not be contacted.

In the final part of this series next month we will look at anti-debugging tricks that target other popular debuggers, as well as some anti-emulating and anti-intercepting tricks.

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. http://www.virusbtn.com/pdf/magazine/2008/200812.pdf.

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

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

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

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

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