TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART FIVE

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 [1–4]) describes some tricks that might become common in the future, along with some countermeasures.

This article will concentrate on anti-debugging tricks that target the *OllyDbg* debugger. All of these techniques were discovered and developed by the author.

1. OllyDbg-specific tricks

OllyDbg is perhaps the most popular of user-mode debuggers. It contains a number of vulnerabilities.

1.1 Malformed files

OllyDbg does not properly support files whose entry point is zero. Zero is a legal starting value for EXE files and allows execution of the MZ header. The file is loaded in *OllyDbg*, but the entry point's break point is not set.

OllyDbg fails to check the values of the Export Address Table Entries field and the Base Relocation Directory Size field prior to performing some arithmetic on them. This can result in an integer overflow and memory corruption.

If the value of the Export Address Table Entries field is 0x40000000 or larger, then *OllyDbg* will start overwriting memory until a crash occurs.

If the value of the Base Relocation Directory Size field is 0x3FFFFFFE or larger, then *OllyDbg* will write the relocated values to unallocated heap memory. On certain platforms, this can result in the execution of arbitrary code. The mitigating factor for the relocation table problem is the fact that it requires a file in excess of one gigabyte in size, because *OllyDbg* reads the relocation data directly from the file.

The Export Address Table Entries and Base Relocation Directory Size bugs affect all versions of *OllyDbg*, including 2.00h. The author of *OllyDbg* released version 2.00h almost 60 days after the bugs were reported, but it still contains the bugs. He has not responded to the report.

1.2 OutputDebugString

OllyDbg passes user-defined data directly to the msvcrt _vsprintf() function. The data can contain formatting string tokens which can cause the _vsprintf() function to access arbitrary memory via the '%s' token. A number of variations of the attack exist, but three tokens are all that is required. The first two tokens can be of any format ('%c', '%x', etc.), but the third token must be a '%s'. This is because the _vsprintf() function calls the __vprinter() function, and passes a zero as the fourth parameter. The fourth parameter is accessed by the third token if the '%s' is used there.

2. OllyDbg plug-ins

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

2.1 antiAnti

antiAnti is hard-coded for a particular language version of *Windows XP SP2* because it uses a constant value for the service table index and for the ntdll NtSetInformationThread() and ntdll NtQueryInformationProcess() functions.

The plug-in patches the debuggee's ntdll NtSetInformationThread() and ntdll NtQueryInformationProcess() function codes so that they simply return. This behaviour is a bug, since the return code is never set.

antiAnti injects a thread into the debuggee's process space, which sets the PEB->BeingDebugged, PEB->NtGlobalFlag and PEB->Heap->ForceFlags flags to zero, and sets the PEB->Heap->Flags flags to 'HEAP_GROWABLE'. The problem with this approach is that if the debuggee's process contains a thread local storage callback, the callback will receive notification of the new thread. It can then query the thread for its entry point and see the injected code.

antiAnti also patches the debuggee's user32 EnableWindow() function code so that it simply returns. This behaviour is another bug, since the return code is never set.

The author of *antiAnti* could not be contacted.

2.2 HideDebugger

HideDebugger changes the address of the kernel32 WaitForDebugEvent() and kernel32 ContinueDebugEvent() functions in *OllyDbg*'s import address table. When a debug event occurs, *HideDebugger* sets the debuggee's PEB->BeingDebugged flag to zero. The problem with this approach is that it can be detected by malware that sets the PEB->BeingDebugged flag to a non-zero value, then calls the kernel32 IsDebuggerPresent() function. If an exception or a debug event occurs (for example, stepping

over the kernel32 IsDebuggerPresent() function call), then *HideDebugger* is revealed because the returned value will be zero.

HideDebugger changes the address of the debuggee's ntdll NtOpenProcess() function in the kernel32.dll's import address table to point to a dynamically allocated block of memory. This block contains code that prevents the *OllyDbg* process ID from being opened, which in turn prevents the *OllyDbg* process from being terminated. This redirected import can easily be identified because it does not point into ntdll.dll's image space.

HideDebugger also changes the address of the debuggee's ntdll NtQueryInformationProcess() function in the kernel32.dll's import address table to point to a dynamically allocated block of memory. This block contains code to watch for queries of the ProcessDebugPort class for the debuggee's process. When one is seen, the handle is set to zero, and then the call is allowed to proceed. This invalid handle will cause an error to be returned. The function should never return an error for the current process handle. Such an error is a sure sign that *HideDebugger* is present. The redirected import can easily be identified because it does not point into ntdll.dll's image space.

Similarly, *HideDebugger* changes the address of the debuggee's ntdll RtlRaiseException() function in the kernel32.dll's import address table to point to a dynamically allocated block of memory. This block contains code to check for the most common form of the kernel32 OutputDebugStringA() function exploit, which is a string that begins with a '%'. This does not solve the general problem, though, because the '%s' tokens can appear anywhere within the string and still cause *OllyDbg* to crash. This redirected import can easily be identified because it does not point into ntdll.dll's image space.

Finally, *HideDebugger* changes the address of the user32 SetWindowTextA() and user32 GetWindowTextA() functions in *OllyDbg*'s import address table. The idea is to prevent *OllyDbg* from changing the caption to include the 'OllyDbg' text. Instead, when the user32 SetWindowTextA() function is called for the *OllyDbg* window, the string is copied into a buffer. When the user32 GetWindowTextA() function is called for the *OllyDbg* window, the cached string is returned.

The author of *HideDebugger* could not be contacted.

2.3 HideOD

HideOD sets the debuggee's PEB->BeingDebugged, PEB->NtGlobalFlag and PEB->Heap->ForceFlags flags to zero, and sets the PEB->Heap->Flags flags to 'HEAP_GROWABLE'.

HideOD sets the debuggee's default heap head and tail values to zero, but that change is visible because the heap chunk sizes are not the expected sizes.

HideOD searches within the debuggee's kernel32 UnhandledExceptionFilter() function for the branch that is evaluated after the ProcessDebugPort is queried. There are two branch types that *HideOD* recognizes, allowing *HideOD* to support *Windows 2000* in the first case, and *Windows XP* and later in the second. There is an earlier branch that could have been patched to achieve the same result, and would have allowed for *Windows NT* support. The branch is overwritten to force the FALSE case, meaning that no debug port exists. This patch is recognizable by the '90' opcode ('NOP' instruction) after the '39' opcode ('CMP' instruction).

HideOD saves the debuggee's original ntdll NtSetInformationThread() function code to a dynamically allocated block of memory, then replaces it with the *Windows XP*-style code: MOV EDX, xxxxxxxx / CALL DWORD PTR DS:[EDX]. This change is instantly recognizable in *Windows NT* or *Windows 2000*, since the code is normally: LEA EDX, DWORD PTR SS:[ESP + 4] / INT 2E. The value that is assigned to EDX is a pointer to the dynamically allocated block of memory. This block intercepts attempts to call the ntdll NtSetInformationThread() function with the HideThreadFromDebugger class, and simply returns success in that case. The bug in this code is that if an invalid handle is passed to the function, then an error code should be returned. A successful return is an indication that *HideOD* is running.

HideOD overwrites *OllyDbg*'s kernel32 OutputDebugStringA() handler function with some code that causes it to return immediately. It appears that something more was intended, because space is allocated for a possible error code, and there is a test for the EIP register being in the upper or lower 2Gb memory space.

HideOD patches the debuggee's kernel32 Process32NextW() function code so that it always returns zero.

HideOD changes a conditional branch into a jump in the debuggee's kernel32 CheckRemoteDebuggerPresent() function. The result is that the function always sets to FALSE the value pointed to by the pbDebuggerPresent argument. The correct behaviour would have been to branch to code that returns FALSE 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.

HideOD saves the debugee's original ntdll NtQueryInformationProcess() function code to a dynamically allocated block of memory, then replaces it with the *Windows XP*-style code: MOV EDX, xxxxxxxx / CALL DWORD PTR DS:[EDX]. This change is instantly recognizable in *Windows NT* or *Windows 2000*. The value that is assigned to EDX is a pointer to the dynamically allocated block of memory. This block intercepts attempts to call the ntdll NtQueryInformationProcess() function with the ProcessDebugPort class, and tries to return zero for the port in that case. However, there is a bug in the code, which does not check if 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, then *HideOD* will cause an exception. *OllyDbg* will trap the exception, but the debugging session will be interrupted. The correct behaviour would have been to call the original handler then zero the port 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 also be taken into account.

The author of *HideOD* could not be contacted.

2.4 IsDebugPresent

IsDebugPresent (also known as *IsDebugExtraHide*) sets the debuggee's PEB->BeingDebugged, PEB->NtGlobalFlag and PEB->Heap->ForceFlags flags to zero. It also runs a thread which can periodically trigger the set after a specified length of time. In the same way as for *HideDebugger*, *IsDebugPresent* can be detected by setting the PEB->BeingDebugged flag to a non-zero value, then calling the IsDebuggerPresent() function after some time. *IsDebugPresent* will be revealed because the returned value will be zero.

The author of *IsDebugPresent* could not be contacted.

2.5 Olly Advanced

Olly Advanced fixes the EXCEPTION_INVALID_HANDLE (0xC0000008) exception 'bug' that occurs when, for example, an invalid handle is passed to the kernel32 CloseHandle() function while a debugger is active. The presence of a debugger causes a debug break to occur, instead of completing the kernel32 CloseHandle() function call. The fix is to patch *OllyDbg* to resume execution and allow the kernel32 CloseHandle() function call to complete as normal.

Olly Advanced forces *OllyDbg* to ignore any failure of the kernel32 TerminateProcess() function.

Olly Advanced hooks the call in *OllyDbg* to the kernel32 CreateProcess() function that creates the process for debugging. When the hook is reached, *Olly Advanced* makes the following changes:

- It searches within the debuggee's ntdll NtQuerySystemInformation() function code for the 'C2' opcode ('RET' instruction) and replaces it with an 'E9' opcode ('JMP' instruction) to point to a dynamically allocated block of memory. There are two problems with this. The first is that the search for the 'C2' opcode is done blindly, so the 'C2' that is seen might be the function index rather than the RET instruction. The second problem is that there might not be five bytes available to replace. If there are fewer than five bytes available, then *Olly Advanced* will destroy part of the following function. This can cause the debuggee to crash randomly.

  The block intercepts attempts to call the ntdll NtQuerySystemInformation() function with the SystemKernelDebuggerInformation class, and tries to return zero for the debugger information in that case. However, there is a bug in that code, which does not check if the SystemInformation parameter points to a valid memory address, or that the entire SystemInformationLength range is writable. If either the SystemInformation pointer or the SystemInformationLength is invalid, then *Olly Advanced* 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 debugger information only if the function returned successfully. It is also unclear why this function is intercepted, since *OllyDbg* is not a kernel-mode debugger.

- *Olly Advanced* performs the same search for the 'C2' opcode ('RET' instruction) within the debuggee's ntdll NtQueryInformationProcess() function code, and once again replaces it with an 'E9' opcode ('JMP' instruction) to point to a dynamically allocated block of memory. The same problems exist here: the search for the 'C2' opcode is done blindly, so the 'C2' that is seen might be the function index rather than the RET instruction, and if there are fewer than five bytes available, then *Olly Advanced* will destroy part of the following function.

  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 UniqueProcessId. However, there is a bug in both codes, which do not check if 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 any reason, then *Olly Advanced* 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. Another problem with the replacement is that the process ID should never be identical to the parent process ID. Not even the EXPLORER.EXE process looks like that. Such a result is an obvious sign that *Olly Advanced* is active.

- *Olly Advanced* performs the same search again within the debuggee's ntdll NtQueryObject() function code, replacing the 'C2' opcode with an 'E9' opcode pointing to a dynamically allocated block of memory. The same problems exist here as described above.

  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 the code, which does not check if 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, then *Olly Advanced* will cause an exception. *OllyDbg* will trap the exception, but the debugging session will be interrupted.

  It would have been better to have zeroed out the entire returned data only if the function returned successfully, but 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 *Olly Advanced* is active.

Olly Advanced hooks the code in *OllyDbg* that is reached when the debuggee's entry point is reached. When the hook is reached, *Olly Advanced* makes the following changes:

- It searches within the debuggee's kernel32 UnhandledExceptionFilter() function code for the code that retrieves the debuggee's PEB pointer. Then it searches for the '0F 84' opcode ('JE' instruction) that follows in order to fix a problem that was introduced in *Windows XP*. The problem is that *Windows XP* doesn't return immediately if the FLG_POOL_ENABLE_TAIL_CHECK (0x100) is set in the debuggee's PEB->NtGlobalFlag flags. *Olly Advanced* patches the branch so that it always returns immediately. However, the result is still that the registered exception handler will not be called because a debugger is present.
- It patches the debuggee's kernel32 Process32NextW(), kernel32 Module32Next(), kernel32 CheckRemoteDebuggerPresent() and kernel32 GetTickCount() function codes, so that they always return zero.
- It hooks the debuggee's ntdll NtSetInformationThread() function by replacing the first five bytes of the function 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.
- It patches the debuggee's kernel32 TerminateProcess() function code to cause it simply to return. This behaviour is a bug, since the return code is never set.
- It sets the debuggee's PEB->BeingDebugged flag to zero.
- It sets the debuggee's PEB->NtGlobalFlag flag to the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value. This is the least incorrect behaviour, and *Olly Advanced* appears to be the only plug-in that does this. However, it is still incomplete.

Olly Advanced hooks the code in *OllyDbg* that is reached when the debuggee's entry point break point is set. When the hook is reached, it searches within *OllyDbg*'s ntdll.dll in-memory image for a particular Thread Local Storage (TLS) callback-specific text string, then searches for the code that references that string. It places a break point in the debuggee's address space at the location after the code that prints the TLS message.

A minor bug exists in the parsing of the MZ header, though, which is that the MZ->lfanew field is assumed to be only 16 bits large. If the PE header is located more than 64kb from the start of the file, then *Olly Advanced* will access an unpredictable memory location while attempting to retrieve the PE->SizeOfImage field value, and then probably crash. However, ntdll.dll is unlikely to have such a large MZ->lfanew field value.

Olly Advanced hooks the code in *OllyDbg* that is reached when the debuggee's PE->BaseOfCode and PE->SizeOfCode fields are cached for later use. When the hook is reached, *Olly Advanced* attempts to calculate the correct sizes for the PE->BaseOfCode and PE->SizeOfCode fields. The MZ->lfanew field size bug is present here.

In this case, the MZ->lfanew field is in the debuggee's executable. We have seen viruses which append a new PE header to the host in order to defeat some heuristic detection methods. Such files can certainly have an MZ->lfanew field value in excess of 64kb. When that happens, *Olly Advanced* will read incorrect data for the PE->COFF->SizeOfOptionalHeader and PE->SizeOfImage fields and the VirtualAddress fields for the first two sections. These unpredictable values could introduce a problem that was not present before.

Olly Advanced forces *OllyDbg* to ignore the debuggee's Export Address Table if the table size appears to be too small or cannot be read completely.

Olly Advanced hooks the code in *OllyDbg* that is reached when *OllyDbg* has finished loading all plug-ins. When the hook is reached, *Olly Advanced* erases the entire Export Address Table and the Export Table Directory. This prevents detection via the kernel32 ReadProcessMemory() function to look for things like the 'ollydbg.exe' DLL name. The MZ->lfanew field size bug is present here, but *OllyDbg* is unlikely to have such a large MZ->lfanew field value.

Olly Advanced forces *OllyDbg* to ignore the data directory/OptionalHeader bug described above.

Olly Advanced hooks the code in *OllyDbg* that is reached when *OllyDbg* is formatting the kernel32 OutputDebugStringA() string. When the hook is reached, *Olly Advanced* checks if the parameter is in readable memory and skips it if not.

Olly Advanced hooks the call in *OllyDbg* to the kernel32 DebugActiveProcess() function. When the hook is reached, *Olly Advanced* touches each page in the first section of the debuggee's ntdll.dll image. The purpose of this is unclear, but it would allow the kernel32 ReadProcessMemory() function to avoid some failures. The MZ->lfanew field size bug is also present here, but ntdll.dll is unlikely to have such a large MZ->lfanew field value.

Olly Advanced sets the PEB->Heap->Flags flags to 'HEAP_GROWABLE', and sets the debuggee's PEB->Heap->ForceFlags flags to zero.

Olly Advanced patches the kernel32 SuspendThread() and user32 BlockInput() function codes to cause them simply to return. This behaviour is a bug, since the return code is never set.

The author of *Olly Advanced* responded to the bug report, saying that the *Olly Advanced* source has been available from the 'VIP' area of the Tuts4you site [5] for some time, but so far no-one has worked on it. As a result, it seems unlikely that the bugs will be fixed.

2.6 OllyICE

OllyICE is a patched version of *OllyDbg*. One of the patches is reached when formatting the kernel32 OutputDebugStringA() string. The patch 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. This is probably the source of the code that is used by the plug-ins *Olly Invisible* and *Olly's Shadow*.

OllyICE ignores both the data directory/OptionalHeader bug and the entry point-zero bug described above.

Another patch is in the __fuistq() function, to avoid the floating-point operations error by changing the value to 9.2233720368547758e+18. That is, the last three digits are removed to keep the value within bounds. However, this fix applies only to the positive value. The negative value will still crash *OllyICE*.

In the penultimate part of this article series (next month) we will look at anti-debugging tricks that target the *OllyDbg* plug-ins *Olly Invisible*, *PhantOm*, *Stealth64* and *Olly Shadow*.

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] Tuts4you. http://forum.tuts4you.com/.