

TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART SEVEN

Peter Ferrie
Microsoft, USA

Unpackers have been around for as long as packers themselves, but anti-unpacking tricks have appeared more recently – and have increased rapidly both in number and, in some cases, complexity.

The final part of this series of articles (see also [1–6]) concentrates on anti-debugging tricks that target a number of popular debuggers, as well as some anti-emulating and anti-intercepting tricks.

All of the techniques described here were discovered and developed by the author.

1. ANTI-DEBUGGING TRICKS

1.1 Immunity Debugger-specific tricks

Immunity Debugger is essentially *OllyDbg* with a Python command-line interface. In fact, large parts of its code are identical, byte for byte, to the *OllyDbg* code. Consequently, it has the same vulnerabilities as *OllyDbg* with respect to both detection and exploitation [5, 6].

1.1.1 Malformed files

Like *OllyDbg*, *Immunity Debugger* 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. Such files are still loaded in *Immunity Debugger*, but in each case the entry point's breakpoint is not set.

Immunity Debugger 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 *Immunity Debugger* will start overwriting memory until a crash occurs.

If the value of the Base Relocation Directory Size field is 0x3FFFFFFE or larger, then *Immunity Debugger* will parse relocations from 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 size of greater than one gigabyte, because *Immunity Debugger* reads the relocation data directly from the file.

The Export Address Table Entries and Base Relocation Directory Size bugs affect all versions of *Immunity Debugger*, including 1.70. The authors of *Immunity Debugger* released version 1.70 more than 60 days after the report was submitted to them. The authors have not responded to the report.

Despite being based on *OllyDbg*, only four of the *OllyDbg* anti-detection plug-ins have been ported to *Immunity Debugger*: *HideDebugger*, *HideOD*, *IsDebugPresent* and *PhantOm*. *IsDebugPresent* is a port of an earlier version, which only sets the debuggee's PEB->BeingDebugged to zero. The others are identical to the *OllyDbg* versions, and thus contain the same bugs [5, 6].

1.1.2 FindWindow

Immunity Debugger can be found by calling the user32 `FindWindow()` function, and then passing 'ID' as the class name to find.

Example code looks like this:

```
push 0
push offset l1
call FindWindowA
test eax, eax
jne being_debugged
...
l1: db "ID", 0
```

1.2 Zeta Debugger-specific tricks

Zeta Debugger is a lesser-known user-mode debugger with a graphical user interface. It supports plug-ins, but so far there are none that hide the presence of the debugger. Its code is very good and does not seem to have any obvious vulnerabilities. However, there is a bug that causes it to crash immediately on *Windows 2000*. The bug relates to the use of the kernel32 `CreateToolhelp32Snapshot()` function on a suspended process. This function was introduced to the *Windows NT*-line in *Windows 2000*, though it existed as far back as *Windows 95* in a separate DLL. On *Windows 2000* and later, it calls into the ntdll `RtlQueryProcessDebugInformation()` function, which performs the majority of the work. Part of that work includes inserting into the process a thread which gathers information about the process. This has the unintended consequence of resuming the process. Since the debugger has attached to the process, *Windows* also creates another thread that executes a breakpoint on behalf of the debugger. The problem is that when the process wakes up, the debug breakpoint will be executed before the debugger can call `WaitForDebugEvent()` to intercept it. Typically, the process would crash at this point. However, there are ways to continue execution and the process will not be under the debugger's control.

Windows XP and later attempt to read from the process memory first. This attempt fails for a suspended process because it has not been completely initialized at that time. As a result, *Windows XP* and later do not create a new thread, so they do not demonstrate the problem.

1.2.1 FindWindow

Zeta Debugger can be found by calling the user32 `FindWindow()` function, and then passing 'Zeta Debugger' as the class name to find.

Example code looks like this:

```
push 0
push offset l1
call FindWindowA
test eax, eax
jne being_debugged
...
11: db "Zeta Debugger", 0
```

1.3 Rock Debugger-specific tricks

Rock Debugger is another less well known user-mode debugger with a graphical user interface. It supports plug-ins, but there are none that hide the presence of the debugger. It does not seem to have any obvious vulnerabilities.

1.3.1 FindWindow

Rock Debugger can be found by calling the user32 `FindWindow()` function, and then passing 'Rock Debugger' as the window name to find.

Example code looks like this:

```
push offset l1
push 0
call FindWindowA
test eax, eax
jne being_debugged
...
11: db "Rock Debugger", 0
```

1.4 Turbo Debug32-specific tricks

Turbo Debug32 used to be a popular debugger for user-mode applications because of its familiar interface and solid performance. However, there are several problems in its code which leave it vulnerable to denial-of-service attacks, unexpected execution points, and even the execution of arbitrary code.

By far the biggest problem in *Turbo Debug32* is the fact that it makes multiple calls to `strcpy()` using stack buffers and user-defined copy sizes. These sizes are not checked before the copy is performed, thus it is possible for an attacker to crash the debugger, or potentially to execute arbitrary code.

Turbo Debug32 attempts to read the entire import table from the process in order to find and hook the kernel32 `ExitProcess()` function. It trusts the Import Table Directory Size field value, and uses it to allocate memory for the import table, regardless of the value that is specified. *Windows* uses the Import Table Directory Size field value as an upper bound value when parsing the import table, not as an allocation size for it. In the case of *Turbo Debug32*, if the size is large enough, the system performance will be impacted severely. Furthermore, since it is possible for the Import Table Directory Size field value to be smaller than the true size of the import table, *Turbo Debug32* might not read enough bytes to parse the import table correctly. As a result, the debugger might attempt to access out-of-bounds memory and crash.

When *Turbo Debug32* is asked to attach to a process that is already running, it assumes that `advapi32.dll` is already present in memory and available to the kernel32 `GetModuleHandle()` function. The correct behaviour would be to call the kernel32 `LoadLibraryA()` function. *Turbo Debug32* calls the kernel32 `GetProcAddress()` function to retrieve the addresses of two functions from `advapi32.dll`, and then calls them without checking if those addresses are non-zero.

When *Turbo Debug32* is asked to step over an instruction, it calculates the length of that instruction, then places a breakpoint at the location of the next instruction. However, the debugger calculates the instruction length for the `0xFF15` opcodes ('CALL' instruction, absolute indirect mode) incorrectly. The calculation code is copied directly from the 16-bit product, which checks for `x6` for absolute addressing (where `x6` represents an instruction encoding where 'x' is any hexadecimal value). However, this is only valid for 16-bit code; in 32-bit code, `x5` is absolute addressing.

Turbo Debug32 also has no understanding of SIB mode. As a result, it writes a `0xCC` opcode ('INT 3' instruction) at the wrong location. This causes a crash in most cases, but it can allow uncontrolled code execution if the new pointer is somewhat valid, and it could be manipulated by an attacker to produce this effect intentionally. It could also be used as a method to detect *Turbo Debug32*.

Example code looks like this:

```
11: call d [offset 13]
12: ...
13: dd offset 12
    db 0cch \
    - (offset $-offset 11) dup (?)
14: dd offset being_debugged
```

By stepping over 11, a breakpoint will be placed inside the 11 instruction, instead of at the location of 12. The effect is

to change the 'call d [offset 13]' instruction into a 'call d [offset 14]' instruction.

1.5 Interactive DisAssembler (IDA)-specific tricks

Interactive DisAssembler, or *IDA*, is the most popular disassembler tool available today. It supports plug-ins.

IDA trusts the value of the Base Relocation Directory Size field, and uses it to allocate memory for the relocation table. However, the relocation table itself may specify a smaller size, because *Windows* uses the Base Relocation Directory Size field value as an upper bound value when parsing the relocation table, not as an allocation size for it. If the Base Relocation Directory Size field value is large enough, the system performance will be impacted severely, and *IDA* might exit unexpectedly.

Recent versions of *IDA* have been extended to include a user-mode debugger. The debugger is implemented as a plug-in for *IDA*. It has a couple of limitations.

The first limitation is during the debugging of files with a PE->ImageBase field value of zero. For such files, the *IDA* debugger will display a message that bears little resemblance to the actual problem. Once the file has loaded, all breakpoints are ignored and attempts to single-step will cause the debugger to resume execution without interruption. This technique has since been disclosed publicly [7].

The second limitation is during the debugging of files which contain multiple relocations pointing to the same memory location. *IDA* will not apply all of the relocation items, leading to an incorrect disassembly. There is no way of producing such a file automatically – manual intervention is required, for example by using a tool. The multiple relocation method can also be combined with the ImageBase zero trick. This combination of techniques is used by the Relock virus.

1.6 IDA plug-ins

A number of packers have been written to detect the *IDA* debugger, so the *IDA Stealth* plug-in was written to attempt to hide the debugger from them. The following is a description of the plug-in, with a list of vulnerabilities that could be used to detect it.

1.6.1 IDA Stealth

IDA Stealth sets the PEB->BeingDebugged and PEB->Heap->ForceFlags flags to zero, and clears all but the HEAP_GROWABLE flag in the PEB->Heap->Flags flags. It clears the FLG_HEAP_ENABLE_TAIL_CHECK,

FLG_HEAP_ENABLE_FREE_CHECK and FLG_HEAP_VALIDATE_PARAMETERS bits in the PEB->NtGlobalFlag field. This behaviour is not as bad as setting bits arbitrarily, but it is still incorrect because the value in the PEB->NtGlobalFlag field can be set by a registry key and/or the debuggee [8].

IDA Stealth hooks the debuggee's ntdll NtQuerySystemInformation() function by replacing the first five bytes with a relative jump to an injected DLL. The hook intercepts attempts to call the ntdll NtQuerySystemInformation() function with the SystemKernelDebuggerInformation class. When that occurs, the hook checks if the SystemInformation parameter points to a valid memory address. However, it does not check whether the SystemInformationLength parameter contains a value that is large enough to hold the complete return value. As a result, if the length is too small, then *IDA Stealth* will cause an exception. The *IDA* debugger will trap the exception, but the debugging session will be interrupted.

If the parameters contain valid values, then *IDA Stealth* will store a value that corresponds to the KdDebuggerEnabled flag that has been cleared and the KdDebuggerNotPresent flag that has been set. However, due to an oversight by the author of the plug-in, the hook then calls the original ntdll NtQuerySystemInformation() function, and returns the true value. This fact was probably not noticed by the author of the plug-in because *IDA* is not a kernel-mode debugger, so unless a real kernel debugger was active at the time, the true value would match the fake one.

The hook also checks if the ntdll NtQuerySystemInformation() function has been called with the SystemProcessInformation class. If it has, 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' (the graphical version of *IDA*), and then erases all copies of the name that are found. The hook does not search for 'idaw.exe' (the console version of *IDA*), though.

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. Due to what appears to be another oversight by the author of *IDA Stealth*, this option is mutually exclusive with the 'hide IDA' option.

IDA Stealth hooks the debuggee's ntdll NtQueryInformationProcess() function by replacing

the first five bytes with a relative jump to an injected DLL. The hook intercepts attempts to call the ntdll NtQueryInformationProcess() function with the ProcessDebugPort class. When that occurs, the hook tries to return a zero for the debug port. However, there is a bug in this code, which is that the hook uses a hard-coded buffer length when it calls the original ntdll NtQueryInformationProcess() function. This can allow the function to succeed, even in cases where the ProcessInformationLength is invalid. As a result, passing an invalid length (longer than allowed) will result in a fixed return length (if the ReturnLength has been specified) and a zeroed port instead of an error code, and *IDA Stealth* is revealed.

The hook also checks if the ntdll NtQueryInformationProcess() function has been called with the ProcessBasicInformation class. If it has, then the hook assumes that the caller is requesting information about itself. The hook replaces the parent process ID with that of the shell window in the InheritedFromUniqueProcessId field, without first checking if the requested process ID is that of the current process. This behaviour is incorrect because the debuggee might be inquiring about a different process. This could also be considered a bug, since the true parent might not be the shell. The correct behaviour would be to use the process ID of *IDA*'s parent.

IDA Stealth hooks the debuggee's ntdll NtQueryObject() function by replacing its first five bytes with a relative jump to an injected DLL. The hook intercepts attempts to call the ntdll NtQueryObject() function with the ObjectAllTypesInformation class. When this occurs, the hook calls the original ntdll NtQueryObject() function, then searches within the returned buffer for the 'DebugObject' string. The hook sets the object counts to zero if the DebugObject string is found. There is a minor bug in the method of comparison, which is that it assumes that the name is zero-terminated. While this is currently the case for DebugObject, it is not a requirement, and there are already other objects with names that are not zero-terminated. The correct method would be to use the length field as the number of characters to compare. Of course, the length should be verified first, to avoid false success on substrings or superstrings, depending on which length is chosen for the comparison. A correct implementation is described in [8].

IDA Stealth hooks the debuggee's ntdll NtClose() function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it registers a Structured Exception Handler before calling the original ntdll NtClose() function. The idea is to consume any exception that occurs. However, a debug

event occurs in the debugger before the exception occurs in the debuggee, and that event cannot be prevented by the debugger. The result is that *IDA* will always break by default. Furthermore, this method does not take into account that, in *Windows XP* and later, any Vectored Exception Handlers that the debuggee registers will run before the registered Structured Exception Handler in *IDA Stealth*. Thus, the presence of the debugger can still be detected on those platforms.

The plug-in hooks the debuggee's kernel32 OutputDebugStringA() and kernel32 OutputDebugStringW() functions by replacing the first five bytes of each with a relative jump to an injected DLL. When the hook is reached, it calls the original kernel32 OutputDebugString() function, and then sets the error code.

IDA Stealth tries to hook the debuggee's ntdll NtSetInformationThread() function by replacing its first five bytes with a relative jump to an injected DLL. The hook would intercept attempts to call the ntdll NtSetInformationThread() function with the HideThreadFromDebugger class, and if that were to occur, the hook would ignore the request and return successfully. However, there are two bugs in the code. The first is that the author of *IDA Stealth* mistyped the name of the function, so it is never hooked. The second is that if an invalid handle is passed to the function, an error code should be returned – a successful return would be an indication that the plug-in is running.

IDA Stealth hooks the debuggee's kernel32 SuspendThread() function by replacing the first five bytes with a relative jump to an injected DLL. When the hook is reached, it simply returns failure. This behaviour is a bug because no error code is returned if an invalid handle is specified.

IDA Stealth hooks the debuggee's kernel32 GetTickCount() function by replacing the first five bytes with a relative jump to an injected DLL. When the hook is reached, it returns a tick count that is incremented by a constant value each time it is called, regardless of how much time has passed. The value of the constant depends on the option that is enabled, and is either 1 or 966.

The plug-in hooks the debuggee's user32 BlockInput() function by replacing the first five bytes with a relative jump to an injected DLL. When the hook is reached, it simply returns successfully.

IDA Stealth hooks the debuggee's kernel32 OpenProcess() function 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 this 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. When the hook is reached, it simply returns success. This behaviour is a bug because no error code is returned if an invalid handle is specified.

The plug-in hooks the debugger's ntdll `DbgUiConvertStateChangeStructure()` function, if it is available, by replacing the 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 ntdll `KiUserExceptionDispatcher()` function by replacing the first five bytes with a relative jump to an injected DLL. When the hook is reached, it saves the values of the debug registers to a private memory block, and then clears them in the context structure, before passing the exception to the debuggee's exception handler. Upon return from the debuggee's exception handler, the hook restores the values of the debug registers, and then resumes execution.

IDA Stealth hooks the debuggee's kernel32 `SetThreadContext()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it saves the contents of the debug registers to a private memory location. It clears the bit in the context structure that specifies that the debug registers are present, and then calls the original kernel32 `SetThreadContext()` function. The effect is to cache the requested changes to the debug registers, but to prevent those changes from occurring.

The plug-in hooks the debuggee's kernel32 `GetThreadContext()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it calls the original kernel32 `GetThreadContext()` function, then merges the cached contents of the debug registers with the true values of the rest of the context. The effect is to simulate the requested changes to the debug registers.

The plug-in hooks the debuggee's ntdll `NtYieldExecution()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it calls the original ntdll `NtYieldExecution()` function, then returns successfully.

IDA Stealth hooks the debuggee's user32 `FindWindowA()`, `user32 FindWindowW()`, `user32 FindWindowExA()` and

`user32 FindWindowExW()` functions by replacing the first five bytes of each with a relative jump to an injected DLL. When the hook is reached, it checks if a class name has been specified. If it has not, then the window name will be used. In either case, the hook converts the name to lower case, and to Unicode if a hook was reached for an ANSI function. The hook then searches for the name within a list carried by the DLL. If a match is found, then the hook returns a failure. Otherwise it calls the original function.

The list of class names is as follows:

- idawindow
- tnavbox
- idaview
- tgrzoom

The list of window names is as follows:

- ida
- graph overview
- idc scripts
- disassembly
- program segmentation
- call stack
- general registers
- breakpoint
- structure offsets
- database notepad
- threads
- segment translation
- imports
- desktopform
- function calls
- structures
- strings window
- functions window
- no signature

The problem is that the entire requested string is searched for each of the names in the list, which means that windows will be hidden if they contain words that include any of the strings in the lists. This mostly affects the 'ida' string. For example, a window with the title 'Acrobat Reader - [hidan.pdf]' [9] will not be visible.

IDA Stealth hooks the debuggee's user32 `EnumWindows()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook

is reached, it replaces the callback function pointer on the stack with a new callback function pointer inside the DLL, and then calls the original function. When the new callback function is reached, it retrieves the window name and searches within the list of window names above. If a match is found, then the callback continues the enumeration. If no match is found it calls the original function.

IDA Stealth hooks the debuggee's `ntdll NtTerminateThread()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it simply returns failure. This behaviour is a bug because no error code is returned if an invalid handle is specified. The same happens when the plug-in hooks the debuggee's `ntdll NtTerminateProcess()` function.

IDA Stealth hooks the debuggee's `kernel32 GetVersion()` function by replacing the first five bytes of the function with a relative jump to an injected DLL. When the hook is reached, it returns a constant value that decodes to version 5.1.2600. This corresponds to *Windows XP*.

The plug-in hooks the debuggee's `ntdll RtlGetVersion()` function 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_OSVERSIONINFOW` or `RTL_OSVERSIONINFOEXW` format has been requested. If the `RTL_OSVERSIONINFOW` format is requested, the hook returns version 5.1.1 with a platform ID that corresponds to *Windows 9x/Me* and a description of 'Service Pack 3'. This information contains two bugs. The first is that the build number is not '2600'. In fact, the correct build number is assigned, but to the wrong structure member. The second bug is that the platform ID does not correspond to a *Windows NT*-based platform.

If the `RTL_OSVERSIONINFOEXW` format is requested, the hook returns version 5.1.2600, with a platform ID that corresponds to a *Windows NT*-based platform, and a description of 'Service Pack 3'.

There is a bug in the code if neither format is requested, which is that no error code is returned.

The author of *IDA Stealth* responded to the report very quickly. The bugs were mostly fixed in beta 2. A number of new bugs were introduced in beta 2, but they were fixed in beta 3.

2. ANTI-UNPACKING BY ANTI-EMULATING

An emulator, as referred to within this paper, is a purely software-based environment, most commonly used by anti-malware software. It places the file to execute inside

the environment and watches the execution for particular events of interest.

2.1 Software interrupts

2.1.1 Interrupt 4

When an `EXCEPTION_INTEGER_OVERFLOW` (0xC0000095) exception occurs, the EIP register has already been advanced to the next instruction, so *Windows* tries to rewind the EIP to point to the proper place. The problem is that *Windows* assumes that the exception is caused by a single-byte 'CE' opcode ('INTO' instruction). If the 'CD 04' opcode ('INT 4' instruction) is used to cause the exception, then the EIP will point to the wrong location. The same behaviour can be seen if any prefixes are placed before the 'INTO' instruction. An emulator that does not behave in the same way will be revealed instantly.

2.1.2 Interrupt 0x0D

When a general protection fault (interrupt 0x0D) occurs, *Windows* attempts to determine the cause of the fault in order to supply the appropriate exception code to the handler. The problem is that there are several ways to produce the general protection fault, which can result in very different exception codes.

For example, attempting to execute an instruction that contains too many prefixes yields `EXCEPTION_ILLEGAL_INSTRUCTION` (0xC000001D). The use of the HLT instruction, any of the descriptor table instructions and certain ports, yields `EXCEPTION_PRIVILEGED_INSTRUCTION` (0xC0000096). Other instructions and ports yield `EXCEPTION_ACCESS_VIOLATION` (0xC0000005). As described elsewhere [10], an instruction that contains the value 0xF0 within the first four bytes yields `EXCEPTION_INVALID_LOCK_SEQUENCE` (0xC000001E).

2.1.3 Interrupt 0x2C

In *Windows NT*, interrupt 0x2C formed one half of an event pair with interrupt 0x2B. A client and a server each controlled one half of the pair, with the server using interrupt 0x2B to pass information to the client, and the client using interrupt 0x2C to pass information to the server.

That functionality was removed in *Windows 2000*. Instead, in *Windows 2000* and *Windows XP*, interrupt 0x2B is the user-mode callback interface for `user32.dll`, and interrupt 0x2C returns the `EXCEPTION_NO_EVENT_PAIR` (0xC000014E) in the EAX register. That functionality was changed again in *Windows Server 2003*. Now, in *Windows Server 2003* and *Windows Vista*, interrupt 0x2C is the

DbgRaiseAssertionFailure() macro, and when it is executed *Windows* issues an EXCEPTION_ASSERTION_FAILURE (0xC0000420) via an exception that can be intercepted.

2.2 File-format tricks

Normally, a PE file requires a non-zero section count and corresponding section descriptors to lay out the file in memory. However, as noted in [8], it is possible to have no section table in the file. As a result, it is also possible to specify explicitly that the file contains no sections. That is, to set the PE->NumberOfSections field to zero. Following such a change, it becomes possible to completely remove the section table on all *Windows NT*-based platforms, including *Windows Vista*. As a result of removing the section table, many tools decide that the file is corrupted and not worthy of examination.

3. ANTI-UNPACKING BY ANTI-INTERCEPTING

3.1 W^X interception

Finally, some unpacking tools work by changing the previously writable-executable page attributes to either writable or executable, but not both. These changes can be detected by using timing attacks, such as a timer query around a local memory write.

Example code looks like this:

```
rdtsc
mov ebx, edx
xchg ecx, eax
;hidden page fault because page is not writable
mov b [offset $], 8bh
rdtsc
sub eax, ecx
sbb edx, ebx
jne being_debugged
cmp eax, 500h
jnbe being_debugged
```

In the example code, the assumption is that the code section is both executable and writable. This is tested by querying a timer (RDTSC), saving the result, attempting to write to the code section, then querying the timer again. In a normal environment, the difference between the two timer values would be small. However, in a W^X environment, the write will cause a page fault because the page attributes have been changed to read-only. The servicing of the page fault will take a long time, and so the difference between the timer values will be large.

CONCLUDING REMARKS

As noted throughout this series, new anti-unpacking techniques continue to be developed as the older ones are constantly being defeated. This series of articles has focused on some of the tricks that might become common in the future, along with some countermeasures.

The text of this article 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 – part six. Virus Bulletin, May 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200905.pdf>.
- [7] Souriz's weblog. #773: bug in IDA-Pro [fails to debug zero-based PE]. <http://souriz.wordpress.com/2008/05/14/773-bug-in-ida-pro-fails-to-debug-zero-base-pe/>.
- [8] Ferrie, P. Anti-unpacker tricks. <http://pferrie.tripod.com/papers/unpackers.pdf>.
- [9] Ferrie, P. Hidan and dangerous. Virus Bulletin, March 2007, p.4. <http://www.virusbtn.com/pdf/magazine/2007/200703.pdf>.
- [10] Ferrie, P. Locked and loaded. <http://pferrie.tripod.com/misc/lowlevel1.htm>.