

TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART TEN

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 [1–10]. Now, the series continues with a look at tricks that are specific to debuggers and emulators.

In this article we look at *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 some plug-ins have been written to attempt to hide it from those packers. The following is a description of those plug-ins, along with the vulnerabilities that could be used to detect them.

1.1 MagicHideOllyDbg

MagicHideOllyDbg is a shameless theft of code from the *HideOD* plug-in. All but five of the important routines are byte-for-byte identical to *HideOD*. In fact, the autorun option still calls itself *HideOD*. One routine is different from *HideOD* because it contains an introduced bug. The bug is in the `OutputDebugStringA` code, which has been renamed to `'OutDebugStringA'`. Since no such API exists in *Windows*, the protection against the *OllyDbg* `OutputDebugStringA` exploit is not enabled. Three more routines are different from *HideOD* thanks to a different compiler setting which has produced slightly different, but functionally equivalent code. The other routine is different because the order of the options checking has been changed. In addition, two new routines have been introduced.

One of the newly introduced routines intercepts the `ntdll NtQuerySystemInformation()` function. *MagicHideOllyDbg* saves the debuggee's original `ntdll NtQuerySystemInformation()` 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. That block intercepts any attempt to call the `ntdll NtQuerySystemInformation()` function with the `SystemModuleInformation` class, and simply returns when such a call is made. This behaviour is a bug because no error code is returned.

Further, the block contains code that is specific to the 32-bit version of *Windows Vista*, using a specific service call which uses the `TEB->Wow32Reserved` field. The use of this option causes an error in *OllyDbg* on other platforms.

Interestingly, the block also contains a check for the `SystemInfoBufferSize` being 11 bytes long. If it is, then the block simply returns, as in the case of the `SystemModuleInformation` class. It is unclear what was intended here.

The other routine hides *OllyDbg*'s caption. The block attempts to intercept the routine that displays the '[CPU-' string. It calls the `kernel32 VirtualProtect()` function to write-enable the memory region, but it only sets the read/write attribute and not the executable bit. Then it calls the `user32 SetWindowText()` function, which sends a message to *OllyDbg*. However, since the pages are no longer executable, *OllyDbg* crashes on DEP-enabled systems.

In the event that everything works, the string is changed to '[CPU] [<current time in hexadecimal>] CPU -'.

The author of *MagicHideOllyDbg* was not contacted regarding these bugs.

1.2 Olly Advanced

The *Olly Advanced* plug-in was described in [6]. What follows are the changes from the previous version, and a description of the behaviour that is specific to more recent versions of *Windows*. Note that the majority of the bugs that were documented previously are also present in this version.

Olly Advanced hooks the code in *OllyDbg* that is reached when a single-step exception occurs. The hook calls the `kernel32 ContinueDebugEvent()` function, and changes the state so that *OllyDbg* ignores the exception. The hook exists to work around a 'bug' in *OllyDbg* that surfaces on the 64-bit platform. The problem is that during the process start-up, there are several DLL unload events. *OllyDbg* sets the T flag in response to those events, before resuming execution of the thread. This causes a side effect similar to the exception priority problem that was described in [2]. The proper solution to the problem is not to set the T flag for DLL unload events during process start-up (which ends when the first breakpoint exception occurs).

Olly Advanced uses a new search method during API hooking. It has replaced the 'C2' ('RET' instruction) search with an 'FF12' ('CALL [EDX]') search. This further ties the plug-in to *Windows XP* and later versions (the plug-in also makes use of the kernel32 `DebugActiveProcessStop()` function, which was introduced in *Windows XP*), since *Windows 2000* and earlier use a different format for native calls. The search is still blind, even though the format is constant for the supported platforms. However, the 64-bit version of *Windows Vista* and later versions use yet another format. The result is that the hook will fail on those platforms, whereas it would previously have succeeded. Fortunately, *Olly Advanced* registers a Structured Exception Handler first, so that when the search fails, it will no longer interfere with *OllyDbg*.

Olly Advanced searches within the debuggee's `ntdll NtOpenProcess()` 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. The block tries to refuse attempts to open the *OllyDbg* process. However, there is a bug in this code. The bug is that the wrong parameter is accessed for the comparison, so the correct result is not guaranteed. Specifically, *Olly Advanced* assumes that the `ProcessHandle` parameter will contain the current process ID, but this parameter is intended only to receive the new object handle, not to specify the process ID. Presumably, the author of *Olly Advanced* examined the behaviour of the kernel32 `OpenProcess()` function, which does place the current process ID in this location. However, the kernel32 `OpenProcess()` function also places the current process ID in its correct location within the `ObjectAttributes` structure, and the `ntoskrnl NtOpenProcess()` function examines only the `ObjectAttributes` structure to determine the current process ID. Thus, it is a simple matter to place different values in the two locations, and bypass the check in *Olly Advanced*.

Olly Advanced searches blindly within the debuggee's `ntdll.dll` for some code that is used by the exception dispatcher, and replaces that code with a jump to a dynamically allocated block of memory. That block saves the debug registers before calling the original exception handler, and then restores the debug registers when the handler returns. The problem with this approach is that the handler might not return – in which case the debug registers can be altered without restriction.

Olly Advanced searches up to 256 bytes within the debuggee's kernel32 `UnhandledExceptionFilter()` function code for an 'FF15' opcode ('CALL' instruction, absolute indirect mode) which points to the `ntdll NtQueryInformationProcess()` function. If that opcode is found, then *Olly Advanced* continues searching without

limit from there for the '0F8C' opcode (long form 'JL' instruction). If that opcode is found, then *Olly Advanced* replaces the branch with an 'E9' opcode ('JMP' instruction). This routine fails on *Windows Vista*, because the call to the `ntdll NtQueryInformationProcess()` function appears earlier in memory than the kernel32 `UnhandledExceptionFilter()` function, so it cannot be found. On *Windows 2000*, the branch exists in an entirely different context, and changing it to an unconditional jump can result in the application being terminated.

Olly Advanced can be directed to patch the kernel32 `GetTickCount()` function to always return 0 as before, or it can hook the debuggee's kernel32 `GetTickCount()` function by replacing the first five bytes with a relative jump to a dynamically allocated block of memory. That block returns a value that increments once per call to the block.

Olly Advanced has changed the behaviour of the TLS breakpoint code. Now, it sets a breakpoint on the first instruction of the first callback.

Olly Advanced fixes a bug in *OllyDbg* that occurs when it is asked to scan the object files within a region of memory that exists outside of any module. Normally, such a request will cause *OllyDbg* to crash with a NULL pointer access. Of course, this is also partly an error on the user's part.

Olly Advanced overwrites the entire contents of the debuggee's `ntdll.dll` code section with that of the debugger's `ntdll.dll` code section. The size of the code section is specified by the `VirtualSize` field in the `SectionHeader`. This has the effect of removing any changes that the debuggee might have made in an attempt to prevent a debugger from attaching to the process. However, this technique is detected very easily.

Example code looks like this:

```
push offset 13
call GetModuleHandleA
push offset 14
push eax
call GetProcAddress
push eax
push esp
push 40h ;PAGE_EXECUTE_READWRITE
push 1
push eax
xchg ebx, eax
call VirtualProtect
mov b [ebx], 0c3h
push eax
push esp
xor eax, eax
push eax
```

```

push ebx
push offset l1
push eax
push eax
call CreateThread
...
11: pop  eax
pop  eax
12: cmp  b [eax], 0c3h
je   l2
jmp  being_debugged
13: db  "ntdll", 0
;use a less common API
14: db  "DbgUserBreakPoint", 0

```

Olly Advanced installs a driver that makes the RDTSC instruction illegal when called from ring 3. The driver returns the current value of the control registers, then disables the RDTSC instruction and enables the RDPIC instruction when called from ring 3 instead. By default, the RDPIC instruction is illegal when called from ring 3. As a result, the missing exception can be used to infer the presence of the driver.

Example code looks like this:

```

xor  eax, eax
push offset l1
push d fs:[eax]
mov  fs:[eax], esp
rdpic
jmp  being_debugged
11: ...

```

Olly Advanced installs a second 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 returns a value that increments once per execution of the RDTSC instruction.

The driver is vulnerable to a kernel-mode crash (blue screen) because of a missing check on one of the parameters. The crash occurs if the control code 0x20 is passed with an invalid input buffer pointer.

Example code looks like this:

```

xor  ebx, ebx
push ebx
push ebx
push 3 ;OPEN_EXISTING
push ebx
push ebx
push ebx
push offset l1
call CreateFileA

```

```

push ebx
push 20h
push eax
call DeviceIoControl
...
11: db "\\.\ring0", 0

```

Olly Advanced has added partial support for the heap flags location for *Windows Vista*. Specifically, *Olly Advanced* knows the new location of the Heap->ForceFlags field, but not the new location of the Heap->Flags field.

The author of *Olly Advanced* did not respond to the report.

1.3 OllyDRX

OllyDRX hooks the code in *OllyDbg* that is reached when *OllyDbg* is 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 bug is the same as that which exists in *OllyICE*.

OllyDRX 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 above. This is the same technique as that used by *Olly's Shadow*.

OllyDRX protects against the public fld bug by altering the value to 9.2233720368547758e+18. That is, the last three digits are removed to keep the value within bounds. This is the same technique as used by *OllyICE*, but the implementation is slightly different. However, as with *OllyICE*, this fix applies only to the positive value. The negative value will still crash *OllyDRX*.

OllyDRX changes the '[E]BX' register string in *OllyDbg* to '[X]XX'; it changes many of the 'CPU' strings to 'DRX'; it changes the 'olly' strings to 'derox'; it changes the 'OllyDbg' strings to 'OllXDRX'; and it changes the 'ODBG' strings to 'DRXG'.

1.4 PhantOm

The *PhantOm* plug-in was described in [7]. What follows are the changes from the previous version.

PhantOm has moved the ntdll NtQueryInformationProcess() function hook into its driver.

The kernel32 BlockInput() function bug has been fixed, however the behaviour remains incorrect. *Windows* will not allow the input to be blocked twice, nor will it allow the input to be enabled twice. Thus, if the same state is passed to the function twice, the result should be different.

Example code looks like this:

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

The *OlyDbg* __fuistq() bug has been fixed in a new way, by replacing the faulting fistp instruction with a fisttp instruction (Floating-point Integer STore with Truncation and Pop). However, this instruction requires a CPU that supports the SSE3 instruction set, otherwise an unsupported opcode exception occurs. *PhantOm* makes no attempt to determine if the instruction is available. This is really not the proper way to solve the problem. As noted in [7], the correct 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 installs a driver which hooks several functions and exposes an interface that can be used to query certain information. The driver is vulnerable to a kernel-mode crash (blue screen) because of missing checks on the output buffer parameter. A crash occurs if the control code 0x30 is passed with an output buffer that begins in a readable page and ends in a non-readable page. The required number of bytes depends on the contents of the output buffer. The crash occurs because the driver ignores the length of the output buffer and assumes that the buffer is entirely readable, for however many bytes it wants to read. By default, four bytes are always read, in order to determine the requested function ('ADD', 'REM' (remove), 'INIT', or 'UNHO' (unhook)). Therefore, the crash can be demonstrated using a buffer that is only one byte large.

Example code looks like this:

```
xor ebx, ebx
push ebx
push ebx
push 3 ;OPEN_EXISTING
push ebx
push ebx
push ebx
push offset 11
```

```
call CreateFileA
push ebx
push ebx
push 1
push offset 12
push ebx
push ebx
push 30h
push eax
call DeviceIoControl
...
;default name, user-configurable
11: db "\\.\extrem", 0
12: db 0 ;place at last byte in page
```

When the ntoskrnl NtQueryInformationProcess() function is called, the hook calls the original ntoskrnl NtQueryInformationProcess() function, and exits if an error occurs, or if the calling process is on the 'allow' list. If no error occurs, then the hook checks the ProcessInformationClass parameter. If the ProcessDebugObjectHandle class is specified, then the hook zeroes the handle. If the ProcessDebugFlags class is specified, then the hook sets the flags to true, signifying that no debugger is present.

The hook also checks if the class 0x23 is specified. This appears to be a bug, and most likely code that was copied accidentally from the ntoskrnl NtQuerySystemInformation() function hook. That class has no relevance in the ntoskrnl NtQueryInformationProcess() context, but it does in the ntoskrnl NtQuerySystemInformation() context.

When the ntoskrnl NtQueryInformationThread() function is called, the hook calls the original ntoskrnl NtQueryInformationThread() function, but forgets to check the result. This makes the hook vulnerable to several kernel-mode crashes, such as when the buffer pointer parameter is null.

Example code looks like this:

```
xor eax, eax
push eax
push eax
push eax
push eax
push eax
push eax
call NtQueryInformationThread
```

The hook intercepts attempts to call the ntoskrnl NtQueryInformationThread() function with the ThreadBasicInformation class if the calling process is not on the 'allow' list. When the ThreadBasicInformation class is seen, the hook tries to zero the returned information and return an error. However, there are two bugs in that code. The first is that the hook does not check if the

ThreadInformation parameter points to a valid memory address, or that the entire range is writable (the hook uses a hard-coded range value). If either the ThreadInformation pointer is invalid for some reason, or the buffer is too small, then *PhantOm* will cause a kernel-mode crash. The second bug is that the hook does not check if the ReturnLength parameter points to a valid memory address (only that it is non-zero), or that all four bytes are writable. If either the ReturnLength pointer is invalid for some reason, or the buffer is too small, then *PhantOm* will cause a kernel-mode crash.

When the NtOpenProcess() function is called, the hook checks if the calling process is on the 'allow' list. If it is not on the list, then the hook tries to check if the ClientID parameter matches that of *OllyDbg* or *CSRSS.EXE*, and then return an error if that is the case. However, there is a bug in that code regarding the validation of the ClientID pointer parameter. The problem is that the hook uses the ntoskrnl MmIsAddressValid() function to check if the page is accessible. This function detects if the address begins within a readable page, but it does not accept any size information, so it can return no information about the page in which the address ends. It is not equivalent to the kernel32 IsBadReadPtr() function. The closest equivalent is the ntoskrnl ProbeForRead() function. As a result, if the pointer begins on a readable page but spans a non-readable page, then *PhantOm* will cause a kernel-mode crash.

Example code looks like this:

```
xor  eax, eax
push offset 11
push  eax
push  eax
push  eax
call  NtOpenProcess
...
11: db 0 ;place at last byte in page
```

When the NtSetContextThread() function is called, the hook tries to clear the CONTEXT_DEBUG_REGISTERS flag from the ContextFlags field, before completing the call. However, there are two bugs in the code. The first relates to the validation of the VirtualAddress pointer parameter. The problem is that the hook uses the ntoskrnl MmIsAddressValid() function to check if the page is accessible. This function detects if the address exists within a readable page, but the function does not accept any access information, so it can return no information about whether or not the page is writable. It is not equivalent to the kernel32 IsBadWritePtr() function. The closest equivalent is the ntoskrnl ProbeForWrite() function. As a result, if the pointer points into a read-only page, then *PhantOm* will cause a kernel-mode crash.

Example code looks like this:

```
push offset 11
push  eax
call  NtSetContextThread
...
11: ;in read-only page
```

The second bug is that the hook applies the change without checking the thread handle. The correct behaviour would be to clear the flag only if the thread belongs to the current process.

The ntoskrnl NtSetInformationThread() function hook now checks for a valid handle by calling the ntoskrnl NtQueryObject() function.

When the ntoskrnl NtQueryObject() function is called, the hook calls the original ntoskrnl NtQueryObject() function, and exits if an error occurs, or if the calling process is on the 'allow' list. If no error occurs, then the hook checks the ObjectInformationClass parameter. If the ObjectTypeInformation class is specified, then the hook zeroes the entire buffer. If the ObjectAllTypesInformation class is specified, then the hook finds the DebugObject object and zeroes the TotalNumberOfHandles field. The routine that is used here was taken from a previous paper [11]. Perhaps as a result of that use, the TotalNumberOfObjects field is left untouched (the code in the previous paper did not examine it, since there was no need), and this fact can be used to detect the presence of *PhantOm*.

Example code looks like this:

```
xor  ebx, ebx
push ebx
push esp ;ReturnLength
;ObjectInformationlength of 0
;to receive required size
push ebx
push ebx
;ObjectAllTypesInformation
push 3
pus  ebx
call  NtQueryObject
pop  ebp
push 4 ;PAGE_READWRITE
push 1000h ;MEM_COMMIT
push  ebp
push  ebx
call  VirtualAlloc
push  ebx
;ObjectInformationLength
push  ebp
push  eax
```

```

;ObjectAllTypesInformation
push 3
push ebx
xchg esi, eax
call NtQueryObject
lods ;handle count
xchg ecx, eax
11: lods ;string lengths
movzx edx, ax ;length
;pointer to TypeName
lods
xch esi, eax
;sizeof(L"DebugObject")
;avoids superstrings
;like "DebugObjective"
cmp edx, 16h
jne 12
xchg ecx, edx
mov edi, offset 13
repe cmpsb
xchg ecx, edx
jne 12
;TotalNumberOfObjects
cmp [eax+4], edx
jne being_debugged
;point to trailing null
12: add esi, edx
;round down to dword
and esi, -4
;skip trailing null
;and any alignment bytes
lods
loop 11
...
13: dw "D","e","b","u","g"
dw "O","b","j","e","c","t"

```

Note that in *Windows Vista* and later versions, this code will fail because the `ntdll NtQueryObject()` function on those platforms does not return the required length when called with an empty buffer (specifically, the 32-bit version of *Windows Vista* returns the wrong length, and the 64-bit version of *Windows Vista* returns no length). Instead, it is required to pass a valid initial buffer, and to increase the size of that buffer until the function stops returning an error.

Example *Windows Vista*-compatible code looks like this:

```

xor ebx, ebx
xor ebp, ebp
xor esi, esi
jmp 12
11: push 8000h ;MEM_RELEASE

```

```

push ebx
push esi
call VirtualFree
12: xor eax, eax
mov ah, 10h ;MEM_COMMIT
add ebp, eax ;4kb increments
push 4 ;PAGE_READWRITE
push eax
push ebp
pus ebx
call VirtualAlloc
;function does not return
;required length for this class
push ebx
;must calculate by brute-force
push ebp
push eax
;ObjectAllTypesInformation
push 3
push ebx
xchg esi, eax
call NtQueryObject
;should check for
;STATUS_INFO_LENGTH_MISMATCH
;but Vista64-incompatible
test eax, eax
jne 11
lods ;handle count
xchg ecx, eax
13: lods ;string lengths
movzx edx, ax ;length
;pointer to TypeName
lods
xchg esi, eax
;sizeof(L"DebugObject")
;avoids superstrings
;like "DebugObjective"
cmp edx, 16h
jne 14
xchg ecx, edx
mov edi, offset 15
repe cmpsb
xchg ecx, edx
jne 14
;TotalNumberOfObjects
cmp [eax+4], edx
jne being_debugged
;point to trailing null
14: add esi, edx
;round down to dword
and esi, -4
;skip trailing null

```

```

;and any alignment bytes
lodsd
loop 13
...
15: dw  "D","e","b","u","g"
dw  "O","b","j","e","c","t"

```

As noted above, there is a small problem regarding the status that is returned by the ntdll NtQueryObject() function, which is that on the 64-bit version of *Windows Vista*, a STATUS_ACCESS_VIOLATION is returned if the buffer is too small, instead of a STATUS_INFO_LENGTH_MISMATCH as for all other platforms. This is why the code does not check for an exact status.

When the ntoskrnl NtQuerySystemInformation() function is called, the hook calls the original ntoskrnl NtQuerySystemInformation () function and exits if an error occurs, or if the calling process is on the 'allow' list. If no error occurs, then the hook checks the SystemInformationClass parameter. If the SystemProcessInformation class is specified, then the hook searches within the returned process list for all processes with a process ID that matches that of *OllyDbg*. If any are found, then the block adjusts the list to skip those entries, and zeroes their contents.

If the SystemKernelDebuggerInformation class is specified, then the hook zeroes the entire buffer. Unfortunately, the interpretation of the returned information is the reverse of what one might expect. A zero in the second byte means that a debugger is *present*. Further, the kernel alters only two bytes, regardless of the buffer size, but *PhantOm* zeroes the entire buffer according to the BufferLength parameter, thus revealing its presence.

Example code looks like this:

```

mov  ebx, offset 11
push 0
push 3
push ebx
;SystemKernelDebuggerInformation
push 23h
call NtQuerySystemInformation
cmp  [ebx+2], al
je   being_debugged
...
11: db  0, 0, 1

```

If the SystemHandleInformation class is specified, then the hook checks each returned entry for a process ID that matches that of *OllyDbg*. If one is found, the hook replaces that process ID with the process ID of Explorer.exe.

The author of *PhantOm* did not respond to the report.

The next part of this series will look at two more *OllyDbg* plug-ins as well as 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] Ferrie, P. Anti-unpacker tricks. <http://pferrie.tripod.com/papers/unpackers.pdf>.
- [2] Ferrie, P. Anti-unpacker tricks – part one. *Virus Bulletin*, December 2008, p.4. <http://www.virusbtn.com/pdf/magazine/2008/200812.pdf>.
- [3] Ferrie, P. Anti-unpacker tricks – part two. *Virus Bulletin*, January 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200901.pdf>.
- [4] Ferrie, P. Anti-unpacker tricks – part three. *Virus Bulletin*, February 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200902.pdf>.
- [5] Ferrie, P. Anti-unpacker tricks – part four. *Virus Bulletin*, March 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200903.pdf>.
- [6] Ferrie, P. Anti-unpacker tricks – part five. *Virus Bulletin*, April 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200904.pdf>.
- [7] Ferrie, P. Anti-unpacker tricks – part six. *Virus Bulletin*, May 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200905.pdf>.
- [8] Ferrie, P. Anti-unpacker tricks – part seven. *Virus Bulletin*, June 2009, p.4. <http://www.virusbtn.com/pdf/magazine/2009/200906.pdf>.
- [9] Ferrie, P. Anti-unpacker tricks – part eight. *Virus Bulletin*, May 2010, p.4. <http://www.virusbtn.com/pdf/magazine/2010/201005.pdf>.
- [10] Ferrie, P. Anti-unpacker tricks – part nine. *Virus Bulletin*, May 2010, p.4. <http://www.virusbtn.com/pdf/magazine/2010/201006.pdf>.
- [11] RtlQueryProcessDebugInformation as Anti-Dbg Trick. Evilcodecave's Weblog. <http://evilcodecave.wordpress.com/2009/04/11/rtlqueryprocessdebuginformation-as-anti-dbg-trick/>.