

# TECHNICAL FEATURE

## ANTI-UNPACKER TRICKS – PART NINE

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 [1–9]. Now, the series continues with a look at tricks that are specific to debuggers and emulators.

In this article we look at anti-debugging tricks including self-modifying code, selectors, RDTSC and *Syser* plug-ins.

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

### 1. Self-modifying code

If a debugger uses the common ‘CC’ opcode (short-form ‘INT 3’ instruction) to place breakpoints during step-over, then it is vulnerable to self-modifying code that removes the breakpoint. As a result, the debugger’s control of the process will be lost. Example code looks like this:

```
mov al, 90h
xor ecx, ecx
inc ecx
mov edi, offset 11
rep stosb
11: nop
```

Of course, there are a couple of variations, such as using ‘rep movs’ instead of ‘rep stos’. The direction flag can be involved, too, in such a way that at a glance, the overwrite might be overlooked. Example code looks like this:

```
mov al, 90h
push 2
pop ecx
mov edi, offset 11
std
rep stosb
nop
11: nop
```

As noted in a previous paper [1], single-stepping is also vulnerable to a variation of this technique, if the overwrite includes the string instruction itself.

The solution to this problem is to use hardware breakpoints instead, though this workaround has its own set of problems. What is not immediately obvious in this example is that the debugger has no way of knowing if the breakpoint that it places at a location is the one that is executed. If the application removes the breakpoint, it can restore it afterwards, and then jump to the address to execute that

breakpoint. The debugger will see the breakpoint exception that it was expecting, and behave as normal. Example code looks like this:

```
mov al, 90h
11: xor ecx, ecx
inc ecx
mov edi, offset 13
12: rep stosb
13: nop
cmp al, 0cch
14: mov al, 0cch
jne 11
15: ...
```

In this example, stepping over the instruction at 12 will allow the code to reach 14. This will cause the breakpoint to be replaced by 12 and executed by 13. The debugger will then regain control. At that time, the only obvious difference will be that the AL register will hold the value 0xCC instead of 0x90, which will allow 15 to be reached in what appears to be one pass instead of two. Of course, much more subtle variations are possible, including the execution of entirely different code-paths.

A variation of the technique can be used as a simple method to detect the presence of a debugger. Example code looks like this:

```
xor ecx, ecx
inc ecx
mov esi, offset 11
lea edi, [esi + 1]
rep movsb
11: mov al, 90h
cmp al, 0cch
je being_debugged
```

### 2. Selectors

Selector values look stable, but they are actually volatile. Specifically, a selector value can be set within a thread, but it might not hold its value for very long. Certain events will cause the value to be changed back. One such event is an exception. In the context of a debugger, the single-step exception can cause some unexpected behaviour. Example code looks like this:

```
xor eax, eax
push fs
pop ds
11: xchg [eax], cl
xchg [eax], cl
```

Single-stepping through this code will cause an access violation exception at 11 because the DS selector will be restored to its default value even before 11 is reached.

A variation of this technique detects the single-step event in a less obvious fashion, simply by checking if the assignment was successful. Example code looks like this:

```

push 3
pop gs
mov ax, gs
cmp al, 3
jne being_debugged

```

This technique is used by Zlob. However, this code is vulnerable to a race condition caused by a thread-switch event, because a thread-switch event also results in the selectors being restored to their default values.

A variation of this technique waits intentionally for a thread-switch event to occur, in order to trigger the effect. Example code looks like this:

```

push 3
pop gs
11: mov ax, gs
shr ax, 1
jb 11

```

This technique is used by Zlob. The code expects the GS selector to become zero again when a thread-switch occurs. This technique works only on the 32-bit versions of *Windows*. It is invalid for 64-bit versions of *Windows* because the GS value on those platforms has bit 0 set, so the loop never exits.

This technique is actually a variation of an anti-emulation trick first seen in 2000, which has been rediscovered. At that time, selectors were not well-supported, so assignments often misbehaved or were ignored completely. Example code looks like this:

```

mov eax, ds
xor ebx, ebx
mov ds, bx
mov ecx, ds
cmp ecx, eax
;detect selector not updated
je being_debugged

```

This technique was first used by Moridin, but Moridin simply checked if the selector held its value when assigned the same value. Example code looks like this:

```

mov edi, ds
push edi
pop ds
mov eax, ds
cmp edi, eax
jnz being_debugged

```

### 3. RDTSC

When the system is powered-on, a timer starts to run, whose value can be queried by the RDTSC instruction. Given how long a typical system takes to boot, and how long it takes for an arbitrary application to be launched, the value that is returned by the RDTSC instruction should be at least *x*, where *x* is a quite large value. Unfortunately, it is common for some hiding tools to intercept the RDTSC instruction,

and to return a small incrementing value instead, which reveals their presence. It might be better to begin with the real value, but there is a problem with that, too. The problem is that if code knows that it is running at start-up, then too large a value reveals that it was not executed in the usual way. This technique could be used to determine the execution state, instead of checking from which directory the application was launched, for example.

### 4. Data-Execution Prevention (DEP)

Data-Execution Prevention is intended to disallow the execution of code from pages that are not marked explicitly as executable. However, for compatibility reasons, the protection is not as secure as it sounds. If code begins in an executable section, and jumps into a non-executable section with DEP enabled, then an exception will occur as expected. However, if execution begins in a non-executable section, then the file will run with DEP silently disabled. This is true even if the code jumps into an executable section, and then back into a non-executable section.

*Turbo Debug32* (and possibly other debuggers) allows breakpoints to be executed in non-executable pages, even in cases where the execution of any other instructions would cause a DEP exception.

### 5. Syser plug-ins

Some packers have been written to detect *Syser*, so a plug-in (only one so far) has been written to attempt to hide *Syser* from those packers. The following is a description of that plug-in, along with its very serious bug.

#### 5.1 HideSyser

*HideSyser* hooks the `ntoskrnl NtCreateFile()` function by overwriting the first five bytes of the handler to point to the driver code, and patching one byte at a fixed offset within the routine. The plug-in works only on *Windows XP*. When run on any other platform, *HideSyser* will cause a kernel-mode crash (blue screen).

The crash is caused by the one-byte patch, which is intended to disable the popping of a frame pointer. This disassembly shows more:

```

mov edi, edi
push ebp
mov ebp, esp
mov edx, [ebp+10]
...
push ebx
push esi
push edi
...
push d [ebp+30]
push d [ebp+2c]

```

```

push d [ebp+28]
push d [ebp+24]
push d [ebp+20]
push d [ebp+1c]
push d [ebp+18]
push d [ebp+14]
push edx
push d [ebp+C]
push d [ebp+8]
call ntcreatefileplus5
pop edi
pop esi
pop ebx
pop ebp
ret 2Ch

```

As we can see, the call to the original `ntoskrnl NtCreateFile()` function intends to use the stack frame that *HideSyser* creates, and if the frame were popped, then the stack would be unbalanced.

However, the patch can be made completely unnecessary by changing the way in which the original `ntoskrnl NtCreateFile()` function is called. Example code looks like this:

```

mov edi, edi
push ebp
mov ebp, esp
...
xor eax, eax
jmp ntcreatefileplus5

```

This allows the API to use the original caller's parameters, thus avoiding the need to push them again. The 'xor eax, eax' line is required to support *Windows NT4* and *Windows 2000*. As a result, this code would work on all versions of *Windows*.

When running on *Windows XP*, the driver code checks for the names '\Device\Syser', '\Device\SyserBoot', '\Device\SyserDbgMsg', '\\.syser' and '\\?.syser', and returns failure if any of them are matched.

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

## 6. OllyDbg-specific

*OllyDbg* was described in a previous paper [6]. The following is a description of a bug that had been discovered since that paper was published.

### 6.1 Step-Over

When *OllyDbg* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *OllyDbg* allows stepping over only the `CALL`, `REP[[N]E]` <string>, and `LOOP[[N]E]` instructions. However, there is a problem if an address-size override is used. Example code looks like this:

```

xor ebx, ebx
push 40h
mov eax, esp
push 3000h
push esp
push ebx
push eax
push -1 ;GetCurrentProcess()
call NtAllocateVirtualMemory
mov b [ebx], 0c3h
call d [bx+1]
ll: ...

```

*OllyDbg* knows that the instruction can be stepped over, but it is confused by the prefix and so does not place any breakpoint at all. As a result, execution resumes freely from 11. This bug was fixed in *OllyDbg* v2.00.

The next part of this series will concentrate on *OllyDbg* plug-ins.

*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>.