

TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART TWELVE

Peter Ferrie
Microsoft, USA

New anti-unpacking tricks continue to be developed as older ones are constantly being defeated. This series of articles describes some tricks that might become common in the future, along with some countermeasures [1–12].

In this article we look at some anti-unpacking tricks that are specific to a range of debuggers.

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

1. HIDE TOOLZ-SPECIFIC

HideToolz is an application that can hide another process under user control. It uses a driver to perform some of its work. The driver searches blindly within the `ntoskrnl KeAddSystemServiceTable()` function code for a particular instruction using a particular register. This combination is only present in *Windows XP* and later versions, so its presence is an unsafe assumption. The driver hooks many functions.

When the `ntoskrnl NtQueryInformationProcess()` function is called, the hook calls the original `ntoskrnl NtQueryInformationProcess()` function, and then exits if an error occurs. Otherwise, the hook checks the `ProcessInformationClass` parameter. If the `ProcessBasicInformation` class is specified, and if the specified process ID is on the hidden list, then the hook replaces the process ID of the parent process 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 the parent process.

If the `ProcessDebugPort` class is specified, then the hook zeroes the debug port, but without checking the process handle. The correct behaviour would be to zero the port only if the current process is specified.

When the `ntoskrnl NtQuerySystemInformation()` function is called, the hook calls the original `ntoskrnl NtQuerySystemInformation()` function, and then exits if an error occurs. Otherwise, the hook checks the `SystemInformationClass` parameter. If the `SystemProcessInformation` class is specified, then for each entry in the hidden list, the hook replaces the process ID of the parent process with the process ID of `Explorer.exe` in the

InheritedFromUniqueProcessId field. Once again, 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 the parent process. A separate option exists which, for each entry in the hidden list, zeroes the entry in the buffer unless the list is requested by *csrss.exe*, *smss.exe*, or any entry in the hidden list.

If the SystemModuleInformation class is specified, then the hook walks the returned list and deletes any entry that contains the name of the driver by copying the entries that follow it over the top, and then reducing the returned length.

If the SystemHandleInformation class is specified, then the hook walks the returned list and deletes any handle for entries in the hidden list by copying the entries that follow it over the top, and then reducing the returned length. This change occurs unless the list is requested by *csrss.exe*, *smss.exe*, or any entry in the hidden list.

When any of the following ntoskrnl functions are called: NtWriteFile(), NtWriteFileGather(), NtShutdownSystem(), NtRaiseHardError(), NtSetSystemPowerState(), NtInitiatePowerAction(), or if the *csrss* ExitWindowsEx() function is called, the corresponding hook can be directed to ignore the request.

When the ntoskrnl NtSetInformationThread() function is called, the hook checks the ThreadInformationClass parameter. If the HideThreadFromDebugger class is specified, then the hook simply returns success. There is a bug in this code, which is that if an invalid handle is passed to the function, then an error code should be returned. A successful return would be an indication that *HideToolz* is running.

When the ntoskrnl NtClose() function is called, the hook calls the ntoskrnl NtQueryObject() function to verify that the handle is valid. If it is, then the hook calls the ntoskrnl NtClose() function. Otherwise, it returns STATUS_INVALID_HANDLE (0xC0000008). However, disabling the exception in this way, without reference to the 'HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag' registry value, means that the absence of the exception might reveal the presence of *HideToolz*.

When either the ntoskrnl NtOpenProcess() function or the ntoskrnl NtOpenThread() function is called, the hook calls the original ntoskrnl function, and then exits if either an error occurs, or the resulting handle refers to any entry on the hidden list.

When the ntoskrnl NtDuplicateObject() function is called, the hook returns an error if the source handle refers to any entry on the hidden list.

HideToolz exposes a private interface via the ntoskrnl NtTerminateProcess() function. The interface is accessed

by passing a handle that is actually a pointer to a memory block, and specifying a termination status of 0xDFF42AB7. The contents of the memory block must begin with the sequence 0x6B 0xB8 0xEC 0x75 0x47 0x46 0x0B 0xFB. Following that is an index into a table of function pointers. Only the values 0–5 are accepted. Next is a pointer to the input buffer, then the size of the input buffer, a pointer to the output buffer, and finally the size of the output buffer. The output buffer is verified to be writable, and the hook requires that the buffer is in user-mode memory.

The author of *HideToolz* could not be contacted.

2. OBSIDIAN-SPECIFIC

Obsidian is an unusual style of user-mode debugger. It does not attach to a process, but instead uses the kernel32 CreateToolhelp32Snapshot() function, and the kernel32 Thread32First() and Thread32Next() functions to access threads. It uses the kernel32 ReadProcessMemory() and WriteProcessMemory() functions to read and write process memory, including to set and clear breakpoints. Even the style of breakpoint is unusual. Rather than the common 'CC' opcode (short-form 'INT 3' instruction) and the T flag to raise single-step exceptions, *Obsidian* places an 'EBFE' opcode ('JMP \$' instruction) at the desired location, and uses a timer with a 'sufficient' delay to allow the execution to complete.

2.1 FindWindow

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

Example code looks like this:

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

2.2 Escape

Because of the breakpoint style in *Obsidian*, it is vulnerable to self-modifying code which is aware of the format of the breakpoint.

Example code looks like this:

```
mov b [offset l1], 0b0h
l1: mov al, 1
;execution resumes freely here
```

This code also functions as a method to detect *Obsidian*, since the value in the AL register will be altered from 1 to 0xFE if *Obsidian* is running.

Obsidian does not handle exceptions, but this limitation is documented already.

Example code looks like this:

```
xor  eax, eax
push offset l1
push  d fs:[eax]
mov  fs:[eax], esp
int  3
...
l1: ;execution resumes freely here
```

The author of *Obsidian* is investigating the report.

3. UGDBG-SPECIFIC

UGDbg is a user-mode debugger with an interface that is similar to *SoftICE*. It can debug both 32-bit and 64-bit applications. It sets the PEB->BeingDebugged and PEB->NtGlobalFlag flags to zero, and does the same for the debugging-heap tail (0xBAADF00D, 0xFEEEFEEEE), if it is present. *UGDbg* also attempts to set the PEB->Heap->ForceFlags flag to zero, however the location of the ForceFlags fields is different in *Windows Vista* and later versions, so the change fails on that platform.

UGDbg uses hardware breakpoints for both single-into and step-over. As a result, it is not vulnerable to the common step-over attack described in [10], nor to any of the variations described below.

4. ROCK DEBUGGER-SPECIFIC

Rock Debugger was described in a previous paper [8]. What follows is a bug that has been discovered since that paper was published.

4.1 Step-over

When *Rock Debugger* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *Rock Debugger* allows the stepping over of any instruction that can be decoded, and which starts with a REP prefix. This leaves the breakpoint vulnerable to self-modifying code.

Example code looks like this:

```
rep
l1: mov b [offset l1], 90h
l2: nop
```

If a step-over is attempted at l1, then execution will resume freely from l2.

Rock Debugger refuses to disassemble code within 15 bytes of the end of a page, if the following page is not readable. Step-over is also disallowed in such cases. This can make it

difficult to debug certain applications, since it is possible to fit several executable instructions within that space.

The author of *Rock Debugger* responded very quickly to the report. He is considering adding a user-defined option to control the behaviour when stepping over instructions that start with a REP prefix.

5. TURBO DEBUG32-SPECIFIC

Turbo Debug32 was described in a previous paper [6]. What follows are bugs that have been discovered since that paper was published.

5.1 Export table

Turbo Debug32 parses the debuggee's export table to identify the offered symbols. It assumes that an ordinal table always exists, and that the contents are valid, even though the table is not used if exports are exported by ordinal only. *Turbo Debug32* uses the values inside the ordinal table as indexes into a memory block within the *Turbo Debug32* process. The accesses are performed without any bounds checking. As such, by placing sufficiently large values into the table, it is possible to cause *Turbo Debug32* to crash.

5.2 Relocated code

Turbo Debug32 does not place the entrypoint breakpoint correctly if the executable file has been relocated in memory as a result of an invalid requested ImageBase (such as loading to offset zero). However, there is no problem if the image is loaded to a random address as a result of Address Space Layout Randomization.

When a process is started, a debugger typically wants to place a breakpoint at the main entrypoint. There are two common ways to locate this address. The first is to query the PEB->Ldr->InMemoryOrderModuleList->EntryPoint field value. Interestingly, *Microsoft* documentation labels this field as 'unsupported', even though the psapi32.dll uses it. The second way is to wait for the CREATE_PROCESS_DEBUG_EVENT event to occur, and then to query the CREATE_PROCESS_DEBUG_INFO->lpStartAddress field value.

However, there is a problem with the second method. *Windows* has supported the relocation of EXE files since *Windows 2000*. With the introduction of *Windows Vista* and Address Space Layout Randomization, this 'feature' came to be supported officially. As a result, a file can be loaded to an address other than the one that it requested. One case in particular is when the requested address is intentionally invalid, such as zero or greater than 2GB. This causes *Windows* to load the file to 0x10000. The problem is that for such files, the CREATE_PROCESS_DEBUG_

INFO->lpStartAddress field value contains the ‘expected’ (and incorrect) entrypoint value, which is calculated by summing the values from the PE->ImageBase and the PE->AddressOfEntryPoint. A breakpoint that a debugger places at that location will not be hit. If the debugger then resumes the process, the process will run freely. Further, the incorrect entrypoint can be calculated to point to a known-writable memory location. The process can then check for a breakpoint at this location and the presence of the debugger will be revealed. This is the problem in *Turbo Debug32*. Other debuggers, such as *OllyDbg*, handle this situation correctly because they use the first method in one form or another, such as by calling the psapi GetModuleInformation() function, which queries the PEB->Ldr->InMemoryOrderModuleList->EntryPoint, and which contains the correct entrypoint value.

This problem has been documented publicly [13].

5.3 Step-over

When *Turbo Debug32* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *Turbo Debug32* allows stepping over only the CALL, REP[[N]E] <string>, and LOOP[[N]E] instructions. The CALL instruction is of particular interest because, as described in a previous paper [8], it does not support the SIB encoding. In the previous example, the bug was exploited to transfer control to an unexpected location. However, a more effective exploitation of the bug allows code to escape the control of the debugger. The bug occurs because *Turbo Debug32* assumes that any SIB-encoded instruction is six bytes long. Thus, if a CALL instruction followed by a JMP instruction can be encoded in no more than six bytes, then stepping over the CALL instruction will allow the JMP to be reached, after which the execution will resume freely from the destination of the JMP instruction.

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 ds:[1]
jmp short l1
nop ;replaced by int 3
l1: ...
```

Turbo Debug32 has another bug regarding instruction decoding, which is that it does not override the address-size

(0x67) when applied to a long displacement. As above, the bug occurs because *Turbo Debug32* assumes that any SIB-encoded instruction is six bytes long. Thus, if a CALL instruction followed by a JMP instruction can be encoded in no more than six bytes, then stepping over the CALL instruction will allow the JMP to be reached, after which the execution will resume freely from the destination of the JMP instruction. 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+80h]
jmp short l1
nop ;replaced by int 3
l1: ...
```

Turbo Debug32 ignores errors when a write occurs beyond writable memory. This bug can also be exploited to allow execution to resume freely from an arbitrary location, if a step-over is attempted at the end of a page. Example code looks like this:

```
xor ecx, ecx
push offset l1
push d fs:[ecx]
mov fs:[ecx], esp
mov eax, offset l2
mov w [eax], 0fee0h ;loopne $
jmp eax
l1: ;reached if step-over at l2
...
l2: ;place at 2nd-last byte in page
```

6. WINDBG-SPECIFIC

WinDbg was described in two previous papers [1, 3]. What follows is a detection method that has been discovered since those papers were published.

6.1 Step-over

When *WinDbg* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *WinDbg* allows stepping over of the CALL (0x9A, 0xE8 and 0xFF & 38 == 0x10), INT (0xCC, 0xCD and 0xCE), REP[[N]E] <string> (including INS and OUTS), LOOP[[N]E] and BOP (0xC4 0xC4) instructions. The BOP support is especially interesting not least because it is undocumented, but also because *WinDbg* knows something

about the format. Specifically, *WinDbg* knows that the 0x50, 0x52-0x54, 0x57-0x58 and 0x5D indexes are four bytes long. For all other BOP indexes, *WinDbg* knows that the instruction is only three bytes long. This might appear to be vulnerable to a step-over bug, since some of the BOP indexes cause exceptions which the debuggee can intercept. However, *WinDbg* takes care to replace the breakpoint with the original byte value prior to dispatching the exception.

WinDbg also allows the stepping over of unknown instructions. This is achieved by using the single-step exception instead of a breakpoint. *WinDbg* also behaves in this way for instructions which contain redundant prefixes. This leaves *WinDbg* vulnerable to detection via the T flag. Example code looks like this:

```
cs:cs:pushfd
pop  eax
test  ah, 1
jne   being_debugged
```

7. FDBG-SPECIFIC

FDBG is a debugger for the 64-bit platform. It can debug 64-bit executables.

7.1 Step-over

When *FDBG* is asked to step over an instruction, it checks if stepping over the instruction is a meaningful request. *FDBG* allows the stepping over of any instruction (valid or not) which starts with a REP prefix. This leaves the breakpoint vulnerable to self-modifying code. Example code looks like this:

```
rep
11: mov b [offset 11], 90h
12: nop
```

If a step-over is attempted at 11, then execution will resume freely from 12.

FDBG refuses to disassemble code within 253 bytes of the end of a page if the following page is not readable. Step-over is disallowed within 31 bytes of the end of a page if the following page is not readable. This can make it difficult to debug certain applications, since it is possible to fit several executable instructions within that space.

The author of *FDBG* responded quickly to the report. A test version was shared privately, which solves these problems, and also the ‘rep stos’ problem (and its variations) described in the ‘Self-modifying code’ section in [10].

8. TITAN ENGINE

TitanEngine is a tool for reverse-engineering applications. It has a built-in debugger. *TitanEngine* uses breakpoints for

any step-over request, regardless of the instruction which is being stepped over. *TitanEngine* supports three kinds of breakpoint instruction – the ‘CC’ opcode (short-form ‘INT 3’ instruction), ‘CD03’ (long-form ‘INT 3’ instruction), and ‘0F0B’ opcode (‘UD2’ instruction). Because of the breakpoint style in *TitanEngine*, it is vulnerable to self-modifying code which is aware of the format of the breakpoint.

Example code looks like this:

```
mov b [offset 11], 0b0h
11: mov al, 1
;execution resumes freely here
```

This code also functions as a method to detect *TitanEngine* if the breakpoint style is not the ‘CC’ opcode form, since the value in the AL register will be altered from 1 to either 3 or 0x0B if *TitanEngine* is running.

The author of *TitanEngine* is investigating the report.

The next part of this series will look at *IDA* plug-ins.

The text of this paper was produced without reference to any Microsoft source code or personnel.

REFERENCES

- [1] <http://pferrie.tripod.com/papers/unpackers.pdf>.
- [2] <http://www.virusbtn.com/pdf/magazine/2008/200812.pdf>.
- [3] <http://www.virusbtn.com/pdf/magazine/2009/200901.pdf>.
- [4] <http://www.virusbtn.com/pdf/magazine/2009/200902.pdf>.
- [5] <http://www.virusbtn.com/pdf/magazine/2009/200903.pdf>.
- [6] <http://www.virusbtn.com/pdf/magazine/2009/200904.pdf>.
- [7] <http://www.virusbtn.com/pdf/magazine/2009/200905.pdf>.
- [8] <http://www.virusbtn.com/pdf/magazine/2009/200906.pdf>.
- [9] <http://www.virusbtn.com/pdf/magazine/2010/201005.pdf>.
- [10] <http://www.virusbtn.com/pdf/magazine/2010/201006.pdf>.
- [11] <http://www.virusbtn.com/pdf/magazine/2010/201007.pdf>.
- [12] <http://www.virusbtn.com/pdf/magazine/2010/201008.pdf>.
- [13] <http://pferrie.tripod.com/misc/lowlevel3.htm>.