ANTI-UNPACKER TRICKS – PART ONE

Peter Ferrie
Microsoft, USA

Unpackers have been around for as long as packers themselves, but anti-unpacking tricks are a more recent development. Anti-unpacking tricks have grown quickly both in number and, in some cases, complexity. This paper is an addendum to a paper presented at the CARO workshop in May this year [1], and describes some of the anti-unpacking tricks that have come to light since that paper was published.

INTRODUCTION

Anti-unpacking tricks come in different forms, depending on what kind of unpacker they are intended to attack. Unpackers can be in the form of memory-dumpers, debuggers, or emulators:

- A memory-dumper dumps the process memory of the running process without regard to the code inside it.
- A debugger attaches to the process, allowing single-stepping, or the placing of breakpoints at key locations, in order to stop execution at the right place. The process can then be dumped with more precision than a memory-dumper alone.
- 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.

There are corresponding tricks for each of the above, and they will be discussed separately.

1. ANTI-DUMPING

1.1 Self-unmapping

The data that fills the image space of a process is simply a mapped view of the file. This view can be unmapped using the kernel32 UnmapViewOfFile() function. If the data is moved first to another location, and then any absolute references are adjusted according to the new image base value, then execution can be resumed from the other location. The result is a process that cannot be dumped by ordinary means.

Example code looks like this:

```
   push 0
   call GetModuleHandleA
   mov ebx, [eax+3ch] ;lfanew
   ;SizeOfImage
   mov   ebx, [eax+ebx+50h]
   push  40h ;PAGE_EXECUTE_READWRITE
   push  1000h ;MEM_COMMIT
   push  ebx
   push  0
   xchg  esi, eax
   call  VirtualAlloc
   mov   ecx, ebx
   lea   edi, [eax+offset l1]
   sub   edi, esi
   push  esi
   push  edi
   xchg  edi, eax
   rep   movsb
   jmp   UnmapViewOfFile
l1: ;execution continues here
   ;but in relocated region
   ...
```

1.2 Page redirection

Page redirection is the extreme implementation of the Guard Pages technique that was described in [1], as it applies to *Armadillo*. In the Guard Pages technique, a guard page is used to allow per-page decryption. It could be defeated by touching the pages, one at a time, and then writing those pages to disk, one at a time. Page redirection avoids that weakness by not restoring the pages to their original location. Instead, all accesses are redirected to other locations in memory where the pages now exist. The result is that the kernel32 ReadProcessMemory() function cannot be used to dump the memory remotely, and the kernel WriteFile() function cannot be used to dump the memory locally using the original addresses, because the redirection will not occur. However, there are two methods that can be used to dump the memory. One is to find the addresses of the redirected pages. This is difficult to automate, and there may be further obfuscation of the content, which will interfere with this method. The second method is simply to perform a user-mode copy of the data, using the original addresses, copying it to a dynamically allocated block of memory. The data can then be written directly from that block of memory. This technique is used by BASLR.

2. ANTI-DEBUGGING

2.1 Special APIs

2.1.1 NtYieldExecution

The ntdll NtYieldExecution() function is used to allow the currently running thread to give up the rest of its execution period and allow any other scheduled thread to execute. The function returns an error if no threads are scheduled to execute. When an application is being debugged, the act of single-stepping through the code causes debug events, and as a result the debugger thread is always scheduled to resume execution. This fact can also be used to infer the presence of a debugger, though it can be used unintentionally to infer the presence of a thread that is running with high priority.

Example code looks like this:

```
    push   20h
    pop    ebp
l1: push   0fh
    call   Sleep
    call   NtYieldExecution
    cmp    al, 1
    adc    ebx, ebx
    dec    ebp
    jne    l1
    inc    ebx
    je     being_debugged
```

This technique is used by the Extreme Debugger Detector.

2.1.2 NtSetLdtEntries

Perhaps because the local descriptor table (LDT) is not used by *Windows*, it is generally not supported properly (or at all) by debuggers. As a result, it can be used as a simple anti-debugger technique. Specifically, a new LDT entry can be created, which maps to some code. Then, by performing a far transfer of control (call or jump) to the new LDT entry, the debugger might become lost or refuse to go further.

Example code looks like this:

```
 ;base must be <= PE->ImageBase
 ;but no need for 64kb align
 base equ 12345678h
 ;sel must have bit 2 set
 ;CPU will set bits 0 and 1
 ;even if we don't do it
 sel equ 777h

 xor eax, eax
 pusheax
 pusheax
 pusheax
 ;4k granular, 32-bit
 ;present, DPL3, exec-only code
 ;limit must not touch kernel mem
 ;calculate carefully to use APIs
 push (base and 0ff000000h) \
     + 0c1f800h \
     + ((base shr 10h) and 0ffh)
 push (base shl 10h) + 0ffffh
 push sel
 call NtSetLdtEntries
 ;jmp far sel:l1
 db  0eah
 dd  offset l1 - base
 dw  sel
l1: ;execution continues here
 ;but using LDT selector
 ...
```

Turbo Debug32 fails to disassemble the code inside the LDT range, but execution continues correctly. *OllyDbg* refuses to continue execution within the LDT range. *WinDbg* disassembles the code correctly inside the LDT range, and execution continues correctly. This technique is used by some malware. It was probably based on some inaccurate documentation on the ReactOS site [2], which misplaces the System bit and includes too many bits in the Type field.

2.1.3 NtQueryInformationProcess

The ntdll NtQueryInformationProcess() function has the following parameters: HANDLE ProcessHandle, PROCESSINFOCLASS ProcessInformationClass, PVOID ProcessInformation, ULONG ProcessInformationLength and PULONG ReturnLength. *Windows Vista* supports 45 classes of ProcessInformationClass information (up from 38 classes in *Windows XP*), but so far only four of them have been documented by *Microsoft*. One of these is the ProcessDebugPort. It is possible to query for the existence (but not the value) of the port. The return value is 0xffffffff if the process is being debugged. Internally, the function queries for the non-zero state of the EPROCESS->DebugPort field. A common method for hiding the debugger process from the ProcessDebugPort class is to zero the value, but without checking the process handle for which the presence of the port is being queried. This presents a problem when a debugger is supposed to be present (for example, in *Armadillo*), since a debug port should exist for that process. This problem has been disclosed publicly [3].

Example code looks like this:

```
xor   ebx, ebx
mov   ebp, offset l1
push  ebp
call  GetStartupInfoA
;sizeof(PROCESS_INFORMATION)
sub   esp, 10h
push  esp
push  ebp
push  ebx
push  ebx
push  1 ;DEBUG_PROCESS
push ebx
push  ebx
push  ebx
push  ebx
push  offset l2
call  CreateProcessA
pop   eax
push  eax
mov   ecx, esp
push  0
push  4 ;ProcessInformationLength
push  ecx
push  7 ;ProcessDebugPort
```

```
    push  eax
    call NtQueryInformationProcess
    pop   eax
    test  eax, eax
    je    being_faked
    ...
    ;sizeof(STARTUPINFO)
 l1: db  44h dup (?)
 l2: db  "myfile", 0
```

2.1.4 CloseHandle

As with an invalid handle, if a protected handle is passed to the kernel32 CloseHandle() function (or directly to the ntdll NtClose() function) and no debugger is present, then an error code is returned. However, if a debugger is present, an EXCEPTION_HANDLE_NOT_CLOSABLE (0xC0000235) exception will be raised. This exception can be intercepted by an exception handler, and is an indication that a debugger is running.

Example code looks like this:

```
    xor eax, eax
    push offset being_debugged
    push d fs:[eax]
    mov fs:[eax], esp
    push eax
    push eax
    push 3 ;OPEN_EXISTING
    push eax
    push eax
    push 80000000h ;GENERIC_READ
    push offset l1
    call CreateFileA
    push eax
    ;HANDLE_FLAG_PROTECT_FROM_CLOSE
    push 2
    push -1
    xchg ebx, eax
    call SetHandleInformation
    push ebx
    call CloseHandle
    ...
 l1: db  "myfile", 0
```

Defeating this method is easiest on *Windows XP*, where a FirstHandler Vectored Exception Handler can be registered by the debugger to hide the exception and silently resume execution. Of course, there is the problem of hooking the kernel32 AddVectoredExceptionHandler() function transparently, in order to prevent another handler from registering as the first handler. However, it is still easier than the transparently hooking the ntdll NtClose() function on *Windows NT* and *Windows 2000* in order to register a Structured Exception Handler to hide the exception. This method has been disclosed publicly [4].

2.1.5 NtSystemDebugControl

The ntdll NtSystemDebugControl() function could have been a very nice function for detecting debuggers. It was introduced in *Windows NT*, and its capabilities increased in subsequent versions of *Windows*. It supported a SysDbgQueryModuleInformation command, which was an alternative to the SystemProcessInformation class of the ntdll NtQuerySystemInformation() function. *Windows XP* introduced the SysDbgReadVirtual command, which allowed the reading of virtual memory from anywhere in the system. There were other commands for writing to virtual memory, reading and writing physical memory and MSRs, among others. Alas, in *Windows 2003 SP1* and later, all of these functions were disabled. The functions that remain are for enabling and disabling the kernel debugger, and querying and setting some minor behaviours.

2.1.6 Non-continuable exceptions

When an exception occurs, the flags specify whether it is continuable or not. A continuable exception is one for which the cause can be corrected, and then execution can resume from the location at which the exception occurred. An example of a continuable exception is a memory access violation. The use of such an exception is how user-mode paging is implemented by packers such as *Shrinker*.

A non-continuable exception is one for which, under normal circumstances, the cause cannot be corrected. An example of a non-continuable exception is a division by zero. Any attempt to resume execution after a non-continuable exception is raised will cause *Windows* to issue an EXCEPTION_INVALID_DISPOSITION exception, prior to terminating the application. However, through the use of a breakpoint in the ntdll RtlRaiseException() function, it is possible to interfere with that sequence. Specifically, the context that is saved on the stack prior to the breakpoint exception can be altered to clear the non-continuable flag. Once that is done, execution of the ntdll RtlRaiseException() function can be resumed. At that point, the call to ntdll RtlRaiseException() becomes indistinguishable from an ordinary call. EXCEPTION_INVALID_DISPOSITION is still delivered to the exception handler, but the application is no longer terminated upon return from the handler. Further, the cause can be corrected. In the case of a division by zero, the divisor can be replaced with a non-zero value, and then the division can be attempted again. This technique has been disclosed publicly [5].

2.2 Hardware tricks

Besides the prefetch queue trick that was described in [1], there is another trick that detects single-stepping. It has worked since the earliest of *Intel* CPUs, and was common in DOS, but it still works in all versions of *Windows*. The trick relies on the fact that certain instructions cause all interrupts to be disabled for one instruction. In particular, loading the SS register clears interrupts for one instruction in order to allow the [E]SP register to be without risk of stack corruption. Of course, the [E]SP register does not need to be loaded. Any instruction can follow the load of the SS register. If a debugger is being used to single-step through the code, then the T flag will be set in the EFLAGS image. This is typically not visible because a debugger will clear the image whenever the flags are saved. However, if the debugger cannot gain control in time to intercept the save, then it has no way of hiding the T flag. Specifically, the debugger cannot gain control if all interrupts are disabled.

Example code looks like this:

```
push ss
pop ss
pushfd
test b [esp+1], 1
jne being_debugged
```

This technique is used by the ChupaChu debugger test.

2.3 Device names

Tools that make use of kernel-mode drivers also need a way of communicating with those drivers. A very common method is through the use of named devices. Any success when attempting to open such a device indicates the presence of the driver.

Example code looks like this:

```
    xor  eax, eax
    mov  edi, offset l2
l1: push eax
    push eax
    push 3 ;OPEN_EXISTING
    push eax
    push eax
    push eax
    push edi
    call CreateFileA
    inc  eax
    jne  being_debugged
    or   ecx, -1
    repne scasb
    cmp  [edi], al
    jne  l1
    ...
l2: <array of ASCIIZ strings,
    null to end>
```

Recent lists include the following names:

\\.\SPCommand

\\.\Syser

\\.\SyserBoot

\\.\SyserLanguage

\\.\SyserDbgMsg

These names belong to *Syser*.

2.4 OllyDbg-specific

A potential arbitrary-code-execution vulnerability was reported recently for *OllyDbg* [6]. However, the author of this paper discovered that the report is incorrect with respect to the target. The problem is not in *OllyDbg*, but in the dbghelp.dll that *OllyDbg* carries. This DLL is used by multiple debuggers, including *IDA*, *SoftICE* and *WinDbg*, but not *Turbo Debug32* or *Immunity Debugger*, for example. It is also shipped as part of *Windows* itself. The version of the DLL that *OllyDbg* carries is indeed vulnerable to a stack buffer overflow. The problem was introduced in *Windows XP*, and corrected in *Windows Server 2003*. In *Windows 2000*, where the file was introduced, a fixed-length copy was performed. Since this was unnecessarily large in most cases, and had the potential to cause crashes because of out-of-bounds reads, it was replaced in *Windows XP* with a string copy. However, the length of the string was not verified prior to performing the copy, leading to the buffer overflow. The fix was to use a string copy with a specified maximum length.

There is a little-known problem in *OllyDbg*'s analyser of floating-point instructions, which is caused by having no mask on invalid floating-point operation errors [7]. This allows two special values to cause floating-point errors when converting from double-extended precision to integer. The values are +/- 9.2233720368547758075e18. There is a publicly available demonstration that specifies only the positive value as a candidate, without mentioning that the negative value is also a candidate.

Example code looks like this:

```
    fld t [offset l1]
    ...
l1: dq -1
    dw 403dh
```

This is the public version. The alternative code looks like this:

```
    fld t [offset l1]
    ...
l1: dq -1
    dw 0c03dh
```

2.5 SoftICE Interrupt 0x2D denial of service

When the kernel32 OutputDebugString() function is called and no user-mode debugger is present, eventually the following code is executed:

```
mov eax, [ebp+8] ;service (1)
;pointer to message structure
mov ecx, [ebp+0ch]
mov edx, [ebp+10h] ;unused (0)
int 2dh
```

If *SoftICE* is installed, then the DbgMsg.sys driver is loaded, even if *SoftICE* isn't running. DbgMsg.sys hooks interrupt 0x2D and executes this code when interrupt 0x2D is executed:

```
    push ecx
    push eax
    call l1
    ...
    ;message structure
l1: mov ecx, [esp+8]
    mov eax, [eax+4] ;bug here
```

The read from [eax+4] without checking first if the pointer is valid, leads to a kernel-mode crash (blue screen) if the original ECX register value is invalid.

Example code looks like this:

```
push 1
pop eax
xor ecx, ecx
int 2dh
```

This interrupt 0x2D bug has already been published [8], but without any details about exactly why it happens.

3. ANTI-EMULATING

3.1 Hardware tricks

3.1.1 Exception priority

Given this code:

```
    xor eax, eax
    push offset l1
    push d fs:[eax]
    mov fs:[eax], esp
    push -1
    popfd
    ;force an exception to occur
    mov eax, [eax]
    ;ExceptionRecord
l1: mov eax, [esp+4]
    ;ExceptionCode
    mov eax, [eax]
l2: ...
```

What is the value of EAX when l2 is reached? Perhaps surprisingly, it is not EXCEPTION_ACCESS_VIOLATION (0xC0000005). It is EXCEPTION_SINGLE_STEP (0x8000004). What actually happens is that an EXCEPTION_ACCESS_VIOLATION is raised, but its delivery to the debuggee is delayed. The reason is that the trap flag remains set when the ntdll KiUserExceptionDispatcher() function gains control, and then after the first instruction there is executed, the single step exception is raised and delivered to the debuggee. Upon returning from the debuggee, after dealing with the EXCEPTION_SINGLE_STEP, the EXCEPTION_ACCESS_VIOLATION is delivered to the debuggee. This technique has been disclosed publicly [9].

3.2 Software Interrupts

3.2.1 Interrupt 0x2E

Interrupt 0x2E is an interface for user-mode code to communicate with native kernel-mode APIs. It was introduced in *Windows NT*, and continues to be supported for compatibility reasons. The interface accepts a function index in the EAX register. A bounds check is performed against this index, prior to dispatching the request if it is valid. However, if the index is out of bounds, then *Windows* will return a STATUS_INVALID_SYSTEM_SERVICE (0xC000001C) value in the EAX register. If the index is within bounds, then a bounds check is performed against the parameter pointer in the EDX register. If the parameter pointer is out of bounds, then *Windows* will return a STATUS_ACCESS_VIOLATION (0xC0000005) value in the EAX register. Some emulators do not support this behaviour.

Example code looks like this:

```
or eax, -1
cdq
int 2eh
int 2eh
cmp eax, 0c0000005h
jne being_debugged
```

This technique is used by the ChupaChu debugger test.

CLOSING REMARKS

Anti-unpacking tricks continue to be developed because the older ones are constantly being defeated. In part two of this series next month, we will describe some tricks that might become common in the future, along with some countermeasures.

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.reactos.org/generated/doxygen/df/d37/struct__LDT__ENTRY.html.

[3] http://forum.tuts4you.com/index.php?showtopic=16750.

[4] http://www.nynaeve.net/?p=203.

[5] http://www.openrce.org/blog/view/1085/Non-continuable_exception_trick.

[6] http://www.securityfocus.com/bid/30139/.

[7] http://board.flatassembler.net/topic.php?t=5820.

[8] http://www.piotrbania.com/all/adv/sice-adv.txt.

[9] http://souriz.wordpress.com/2008/05/09/bug-in-olly-windows-behavior-and-peter-ferrie/.