ANTI-UNPACKER TRICKS

CURRENT

Peter Ferrie, *Senior Anti-Virus Researcher, Microsoft Corporation*

Abstract **Unpackers are as old as the packers themselves, but anti-unpacking tricks are a more recent development. These anti-unpacking tricks have developed quickly in number and, in some cases, complexity. In this paper, we will describe some of the most common anti-unpacking tricks, along with some countermeasures.**

INTRODUCTION

Anti-unpacking tricks can come in different forms, depending on what kind of unpacker they want to attack. The unpacker can be in the form of a *memory-dumper*, a *debugger*, an *emulator*, a *code-buffer*, or a *W-X interceptor*. It can be a tool in a virtual machine. There are corresponding tricks for each of these, and they will be discussed separately.

- 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 used 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.

- A *code-buffer* is similar to, but different from, a debugger. It also attaches to a process, but instead of executing instructions in-place, it copies each instruction into a private buffer and executes it from there. It allows fine-grained control over execution as a result. It is also more transparent than a debugger, and faster than an emulator.

- A *W-X interceptor* uses page-level tricks to watch for write-then-execute sequences. Typically, an executable region is marked as read-only and executable, and everything else is marked as read-only and non-executable (or simply non-present, depending on the hardware capabilities). Then the code is allowed to execute freely. The interceptor intercepts exceptions that are triggered by writes to read-only pages, or execution from non-executable or non-present pages. If the hardware supports it, a read-only page will be replaced by a writable but non-executable page, and the write will be allowed to continue. Otherwise, the single-step exception will be used to allow the write to complete, after which the page will be restored to its non-present state. In either case, the page address is kept in a list. In the event of exceptions triggered by execution of non-executable or non-present pages, the page address is compared to the entries in that list. A match indicates the execution of newly-written code, and is a possible host entrypoint.

I. ANTI-UNPACKING BY ANTI-DUMPING

a. SizeOfImage

The simplest of anti-dumping tricks is to change the SizeOfImage value in the Process Environment Block (PEB). This interferes with process access, including preventing a debugger from attaching to the process. It also breaks tools such as *LordPE* in default mode, among others.

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
mov eax, [eax+0ch] ;LdrData
;get InLoadOrderModuleList
mov eax, [eax+0ch]
;adjust SizeOfImage
add dw [eax+20h], 1000h
```

The technique is used by many packers now. However, the technique is easily defeated, even by user-mode code. We can simply ignore the SizeOfImage value, and call the VirtualQuery() function instead. The VirtualQuery() function returns the number of sequential pages whose attributes are the same. Since there cannot be gaps between sections in memory, the ranges can be enumerated by querying the first page after the end of the previous range. The enumeration would begin with the ImageBase page and continue while the MEM_IMAGE type is returned. A page that is not of the MEM_IMAGE type did not come from the file.

b. Erasing the header

Some unpackers examine the section table to gather interesting information about the image. Erasing or altering that section table in the PE header can interfere with the gathering of that information. This is typically used as to defeat *ProcDump*-style tools, which rely on the section table to dump the image.

Example code looks like this:

```
;get image base
push 0
call GetModuleHandleA
push eax
push esp
push 4 ;PAGE_READWRITE
;rounded up to hardware page size
push 1
push eax
xchg edi, eax
call VirtualProtect
xor  ecx, ecx
mov  ch, 10h ;assume 4kb pages
;store VirtualProtect return value
rep  stosb
```

This technique is used by *Yoda's Crypter*, among others. As above, the VirtualQuery() function can be used to recover the image size, and some of the layout (i.e. which pages are executable, which are writable, etc), but there is no way to recover the section table once it has been erased.

c. Nanomites

Nanomites are a more advanced method of anti-dumping. They were introduced in *Armadillo*. They work by replacing branch instructions with an "int 3" instruction, and using tables in the unpacking code to determine the details. The details in this case are whether or not the "int 3" is a nanomite or a debug break; whether or not the branch should be taken, if it is a nanomite; the address of the destination, if the branch is taken; and how large the instruction is, if the branch is not taken.

A process that is protected by nanomites requires self-debugging (known as "Debug Blocker" in *Armadillo*, see Anti-Debugging:Self-Debugging section below), which uses a copy of the same process. This allows the debugger to intercept the exceptions that are generated by the debuggee when the nanomite is hit. When the exception occurs in the debuggee, the debugger recovers the exception address and searches for it in an address table. If a match is found, then the nanomite type is retrieved from a type table. If the CPU flags match the type, then the branch will be taken. When that happens, the destination address is retrieved from a destination table, and execution resumes from that address. Otherwise, the size of the branch is retrieved from the size table, in order to skip the instruction.

d. Stolen Bytes

Stolen bytes are opcodes that are taken from the host and placed in dynamically allocated memory, where they will be executed separately. A jump instruction is placed at the start of the stolen bytes in the host, to point to the start of the relocated code. A jump instruction is placed at the end of the relocated code, to point to the end of the stolen bytes. The rest of the opcodes in the stolen region in the host are then replaced with garbage. The relocated code can also be interspersed with garbage instructions, in order to make it more difficult to determine the real instructions from the fake instructions. This complicates the restoration of the original code. This technique was introduced in *ASProtect*.

e. Guard Pages

Guard pages act as a one-shot access alarm. The first time that a guard page is accessed for any reason, an EXCEPTION_GUARD_PAGE (0x80000001) exception will be raised. This can be used for a variety of things, but overall it acts as a demand-paging system for ring 3 code. The technique is achieved by intercepting the EXCEPTION_GUARD_PAGE (0x80000001) exception, checking if the page is within a particular range (for example, within the process image space), then mapping in some appropriate content if so.

This technique is used by *Shrinker* to perform on-demand decompression. By decompressing only the pages that are accessed, the startup time is reduced significantly. The committed memory consumption can be reduced, since any pages that are not accessed do not need any physical memory to back them. The overall application performance can also be increased, when compared to other packers that decompress the entire application immediately. *Shrinker* works by hooking the ntdll KiUserExceptionDispatcher() function, and watching for the EXCEPTION_GUARD_PAGE (0x80000001) exception. If the exception occurs within the process image space, then *Shrinker* will load from disk the individual page that is being accessed, decompress it, and then resume execution. If an access spans two pages, then upon resuming, an exception will occur for the next page, and *Shrinker* will load and decompress that page, too.

A variation of this technique is used by *Armadillo*, to perform on-demand decryption (known as "CopyMem2"). However, as with nanomites, it requires the use of self-debugging. This is in contrast to *Shrinker*, which is entirely self-contained. *Armadillo* decompresses all of the pages into memory at once, rather than loading them from disk when they are accessed. *Armadillo* uses the debugger to intercept the exceptions in the debuggee, and watches for the EXCEPTION_GUARD_PAGE (0x80000001) exception. If the exception occurs within the process image space, then *Armadillo* will decrypt the individual page that is being accessed, and then resume execution. If an access spans two pages, then upon resuming, an exception will occur for the next page, and *Armadillo* will decrypt that page, too.

If performance were not a concern, a protection method of this type could also remember the last page that was loaded, and discard it when an exception occurs for another page (unless the exception address suggests an access that spanned them). That way, no more than two pages will ever be in the clear in memory at the same time. In fact, that *Armadillo* does not do this could be considered a weakness in the implementation, because by simply touching all of the pages in the image, *Armadillo* will decrypt them all, and then the process can be dumped entirely.

f. Imports

The list of imported functions can be very useful to get at least some idea of what a program does. To combat this, some packers alter the import table after the imports have been resolved. The alteration typically takes the form of completely erasing the import table, but there are variations that include changing the imported address to point to a private buffer that is allocated dynamically. Within the buffer is a jump to the real function address. This buffer is usually not dumped by default, so when the process exits, the information is lost as to the real function addresses.

g. Virtual machines

Virtual machines are perhaps the ultimate in anti-dumping technology, because at no point is the directly executable code ever visible in memory. Further, the import table might contain only the absolutely required functions (LoadLibrary() and GetProcAddress()), leaving no clue as to what the program does. Additionally, the p-code might be encoded in some way, such that two behaviourally identical samples might have very different-looking contents. This technique is used by *VMProtect*.

The p-code itself can be polymorphic, where do-nothing instructions are inserted into the code flow, in the same way as is often done for native code. This technique is used by *Themida*.

The p-code can contain anti-debugging routines, such as checking specific memory locations for specific values (see Anti-Debugging section below). This technique is used by *HyperUnpackMe2*[i].

The p-code interpreter can be obfuscated, such that the method for interpretation is not immediately obvious. This technique is used by *Themida* and *Virtual CPU*[ii].

II. ANTI-UNPACKING BY ANTI-DEBUGGING

a. PEB fields

i. NtGlobalFlag

The NtGlobalFlag field exists at offset 0x68 in the PEB. The value in that field is zero by default. On *Windows 2000* and later, there is a particular value that is typically stored in the field when a debugger is running. The presence of that value is not a reliable indication that a debugger is really running (especially since it is entirely absent on *Windows NT*). However, it is often used for that purpose. The field is composed of a set of flags. The value that suggests the presence of a debugger is composed of the following flags:

FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)

Example *incorrect* code looks like this:

```
mov eax, fs:[30h] ;PEB
;check NtGlobalFlag
cmp b [eax+68h], 70h
jne being_debugged
```

This technique is used by *ExeCryptor*, among others.

The "cmp" instruction above is a common mistake. The assumption is that no other flags can be set, which is not true. Those three flags alone are usually set for a process that is created by a debugger, but not for a process to which a debugger attaches afterwards. However, there are three further exceptions.

The first exception is that additional flags can be set for all processes, by a registry value. The registry value is the "GlobalFlag" string value of the "HKLM\System\CurrentControlSet\Control\Session Manager" registry key.

The second exception is that all of the flags can be controlled on a per-process basis, by a different registry value. The registry value is the also the "GlobalFlag" string value (note that "*Windows* Anti-Debug Reference" by Nicolas Falliere[iii] incorrectly calls it "GlobalFlags") of the "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>" registry key. The "<filename>"

must be replaced by the name of the executable file (not a DLL) to which the flags will be applied when the file is executed. An empty "GlobalFlag" string value will result in no flags being set.

The third exception is that, on *Windows 2000* and later, all of the flags can be controlled on a per-process basis, by the Load Configuration Structure. The Load Configuration Structure has existed since *Windows NT*, but the format was not documented by *Microsoft* in the PE/COFF Specification until 2006 (and incorrectly). The structure was extended to support Safe Exception Handling in *Windows XP*, but it also contains two fields of relevance to this paper: GlobalFlagsClear and GlobalFlagsSet. As their names imply, they can be used to clear and/or set any combination of bits in the PEB->NtGlobalFlag field. The flags specified by the GlobalFlagsClear field are cleared first, then the flags specified by the GlobalFlagsSet field are set. This means that even if all of the flags are specified by the GlobalFlagsClear field, any flags that are specified by the GlobalFlagsSet field will still be set. No current packer supports this structure.

If the FLG_USER_STACK_TRACE_DB (0x1000) is specified to be set, either by the "GlobalFlag" registry value, or in the GlobalFlagsSet field, the FLG_HEAP_VALIDATE_PARAMETERS will automatically be set, even if it is specified in the GlobalFlagsClear field.

Thus, the correct implementation to detect the default value is this one:

```
mov eax, fs:[30h] ;PEB
mov al, [eax+68h] ; NtGlobalFlag
and al, 70h
cmp al, 70h
je  being_debugged
```

The simplest method to defeat this technique is to create the empty "GlobalFlag" string value.

b. Heap flags

The process default heap is another place to find debugging artifacts. The base heap pointer can be retrieved by the kernel32 GetProcessHeap() function. Some packers avoid using the API and look directly at the PEB instead.

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
;get process heap base
mov eax, [eax+18h]
```

Within the heap are two fields of interest. The PEB->NtGlobalFlags field forms the basis for the values in those fields. The first field (Flags) exists at offset 0x0c in the heap, the second one (ForceFlags) is at offset 0x10 in the heap. The Flags field indicates the settings that were used for the current heap block. The ForceFlags field indicates the settings that will be used for subsequent heap manipulation. The value in the first field is two by default, the value in the second field is zero by default. There are particular values that are typically stored in those fields when a debugger is running, but the presence of those values is not a reliable indication that a debugger is really running. However, they are often used for that purpose.

The fields are composed of a set of flags. The value in the first field that suggests the presence of a debugger is composed of the following flags:

HEAP_GROWABLE (2)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
;get process heap base
mov eax, [eax+18h]
mov eax, [eax+0ch] ;Flags
dec eax
dec eax
jne being_debugged
```

The value in the second field that suggests the presence of a debugger is composed of the following flags:

HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
;get process heap base
mov eax, [eax+18h]
cmp [eax+10h], 0 ;ForceFlags
jne being_debugged
```

The "tail" flags are set in the heap fields if the FLG_HEAP_ENABLE_TAIL_CHECK flag is set in the PEB->NtGlobalFlags field. The "free" flags are set in the heap fields if the FLG_HEAP_ENABLE_FREE_CHECK flag is set in the PEB->NtGlobalFlags field. The validation flags are set in the heap fields if the FLG_HEAP_VALIDATE_PARAMETERS flag is set in the PEB->NtGlobalFlags field. However, the heap flags can be controlled on a per-process basis, through the "PageHeapFlags" value, in the same manner as "GlobalFlag" above.

c. The Heap

The problem with simply clearing the heap flags is that the initial heap will have been initialised with the flags active, and that leaves some artifacts that can be detected. Specifically, at the end of the heap block will one definite value, and one possible value. The HEAP_TAIL_CHECKING_ENABLED flag causes the sequence 0xABABABAB to always appear twice at the exact end of the allocated block. The HEAP_FREE_CHECKING_ENABLED flag causes the sequence 0xFEEEFEEE (or a part thereof) to appear if additional bytes are required to fill in the slack space until the next block.

Example code looks like this:

```
mov   eax, <heap ptr>
;get unused_bytes
movzx ecx, b [eax-2]
movzx edx, w [eax-8] ;size
sub   eax, ecx
lea   edi, [edx*8+eax]
mov   al, 0abh
mov   cl, 8
repe  scasb
je    being_debugged
```

These values are checked by *Themida*.

d. Special APIs

i. IsDebuggerPresent

The kernel32 IsDebuggerPresent() function was introduced in *Windows 95*. It returns TRUE if a debugger is present. Internally, it simply returns the value of the PEB->BeingDebugged flag.

Example code looks like this:

```
call IsDebuggerPresent
test al, al
jne  being_debugged
```

Some packers avoid using the kernel32 IsDebuggerPresent() function and look directly at the PEB instead.

Example code looks like this:

```
mov eax, fs:[30h] ;PEB
;check BeingDebugged
cmp b [eax+2], 0
jne being_debugged
```

To defeat these methods requires only setting the PEB->BeingDebugged flag to FALSE. A common convenience while debugging is to place a breakpoint at the first instruction in the kernel32 IsDebuggerPresent() function. Some unpackers check explicitly for this breakpoint.

Example code looks like this:

```
    push offset l1
    call GetModuleHandleA
    push offset l2
    push eax
    call GetProcAddress
    cmp  b [eax], 0cch
    je   being_debugged
    ...
l1: db "kernel32", 0
l2: db "IsDebuggerPresent", 0
```

Some packers check that the first byte in the function is the "64" opcode ("FS:" prefix).

Example code looks like this:

```
    push offset l1
    call GetModuleHandleA
    push offset l2
    push eax
    call GetProcAddress
    cmp  b [eax], 64h
    jne  being_debugged
    ...
l1: db "kernel32", 0
l2: db "IsDebuggerPresent", 0
```

ii. CheckRemoteDebuggerPresent

The kernel32 CheckRemoteDebuggerPresent() function has these parameters: HANDLE hProcess, PBOOL pbDebuggerPresent. The function is a wrapper that was introduced in *Windows XP* SP1, to query a value that has existed since *Windows NT*. "Remote" in this sense refers to a separate process on the same machine. The function sets to 0xffffffff the value to which the pbDebuggerPresent argument points, if a debugger is present. Internally, it simply returns the value from the ntdll NtQueryInformationProcess (ProcessDebugPort class) function.

Example code looks like this:

```
push eax
push esp
push -1 ;GetCurrentProcess()
call CheckRemoteDebuggerPresent
pop  eax
test eax, eax
jne  being_debugged
```

Some packers avoid using the kernel32 CheckRemoteDebuggerPresent() function, and call the ntdll NtQueryInformationProcess() function directly.

iii. NtQueryInformationProcess

The ntdll NtQueryInformationProcess() function has these parameters: HANDLE ProcessHandle, PROCESSINFOCLASS ProcessInformationClass, PVOID ProcessInformation, ULONG ProcessInformationLength, PULONG ReturnLength. *Windows Vista* supports 45 classes of ProcessInformationClass information (up from 38 in *Windows XP*), but only four of them are documented by *Microsoft* so far. One of them is the ProcessDebugPort. It is possible to query for the existence (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.
Example code looks like this:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 7 ;ProcessDebugPort
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
jne  being_debugged
```

This technique is used by *MSLRH*, among others. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

iv. Debug Objects

Windows XP introduced a "debug object". When a debugging session begins, a debug object is created, and a handle is associated with it. It is possible to query for the value of this handle, using the undocumented ProcessDebugObjectHandle class.
Example code looks like this:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
;ProcessDebugObjectHandle
push 1eh
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
jne  being_debugged
```

This technique is used by *HyperUnpackMe2*, among others. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

The undocumented ProcessDebugFlags class returns the inverse value of the EPROCESS->NoDebugInherit bit. That is, the return value is FALSE if a debugger is present.
Example code looks like this:

```
push eax
mov  eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 1fh ;ProcessDebugFlags
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
je   being_debugged
```

This technique is used by *HyperUnpackMe2*, among others. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

Yet another method is the SystemKernelDebuggerInformation class, which is supported by *ReactOS*[iv], but apparently not by any current version of *Windows*.
Example code looks like this:

```
push eax
mov  eax, esp
push 0
push 2 ;ProcessInformationLength
push eax
;SystemKernelDebuggerInformation
push 23h
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test ah, ah
jne  being_debugged
```

This technique is used by *SafeDisc*. Since this information comes from the kernel, there would be no easy way to prevent this call from revealing the presence of the debugger.

v. NtQueryObject

The ntdll NtQueryObject() function has these parameters: HANDLE Handle, OBJECT_INFORMATION_CLASS ObjectInformationClass, PVOID ObjectInformation, ULONG ObjectInformationLength, PULONG ReturnLength. *Windows NT*-based platforms support five classes of ObjectInformationClass information, but only two of them are documented by *Microsoft* so far. Neither of them is the ObjectAllTypesInformation, which we require.

As noted above, when a debugging session begins on *Windows XP*, a debug object is created, and a handle is associated with it. It is possible to query for the list of existing objects, and check the number of debug objects that exist. This API is supported by *Windows NT*-based platforms, but only *Windows XP* and later will return a debug object in the list.

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
    push  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
    lodsd ;handle count
    xchg  ecx, eax
l1: lodsd ;string lengths
    movzx edx, ax ;length
    ;pointer to TypeName
    lodsd
    xchg  esi, eax
    ;sizeof(L"DebugObject")
    ;avoids superstrings
    ;like "DebugObjective"
    cmp   edx, 16h
    jne   l2
    xchg  ecx, edx
    mov   edi, offset l3
    repe  cmpsb
    xchg  ecx, edx
    jne   l2
    ;TotalNumberOfObjects
    cmp   [eax], edx
    jne   being_debugged
    ;point to trailing null
l2: add   esi, edx
    ;round down to dword
    and   esi, -4
    ;skip trailing null
    ;and any alignment bytes
    lodsd
    loop  l1
    ...
l3: dw    "D","e","b","u","g"
    dw    "O","b","j","e","c","t"
```

Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger.

vi. Thread hiding

Windows 2000 introduced an explicitly anti-debugging API extension, in the form of an information class called HideThreadFromDebugger. It can be applied on a per-thread basis, using the ntdll SetInformationThread() function.

Example code looks like this:

```
push 0
push 0
;HideThreadFromDebugger
push 11h
push -2 ;GetCurrentThread()
call NtSetInformationThread
```

When the function is called, the thread will continue to run but a debugger will no longer receive any events related to that thread. Among the missing events are that the process has terminated, if the main thread is the hidden one. This technique is used by *HyperUnpackMe2*, among others.

vii. OpenProcess

When a process acquires the SeDebugPrivilege, it gains full control of the CSRSS.EXE, even though CSRSS.EXE is a system process. The reason for that is because SeDebugPrivilege overrides all of the restrictions for that process alone. Further, the privilege is passed to child processes, such as the ones created by a debugger. The result is if a debugged application can obtain the process ID for CSRSS.EXE, it can open the process via the kernel32 OpenProcess() function. The process ID can be obtained by the kernel32 CreateToolhelp32Snapshot() function and a kernel32 Process32Next() function enumeration; or the ntdll NtQuerySystemInformation (SystemProcessInformation (5)) function (and the ntdll NtQuerySystemInformation() function is how the kernel32 CreateToolhelp32Snapshot() function gets its information on *Windows NT*-based platforms). Alternatively, *Windows XP* introduced the ntdll CsrGetProcessId() function, which simplifies things greatly.

Example code looks like this:

```
call CsrGetProcessId
push eax
push 0
push 1f0fffh ;PROCESS_ALL_ACCESS
call OpenProcess
test eax, eax
jne  being_debugged
```

This opens (no pun intended) the way to a system-level denial-of-service, by causing the CSRSS.EXE process to perform an illegal operation. One method is the creation of a thread at an invalid memory address, or a thread that executes an infinite loop. However, since the control is complete, an application can inject a thread into the CSRSS.EXE process space and perform some meaningful action, which results in a privilege elevation. However, this is of only minor concern, since usually only Administrators will be able to acquire the debug privilege, and Administrators are highly privileged already. This technique was described publicly by Piotr Bania[v] in 2005.

Both *OllyDbg* and *WinDbg* acquire the debug privilege, but *Turbo Debug* does not. The best way to defeat this technique is to not acquire the privilege arbitrarily, and keep it for only as long as truly necessary.

viii. CloseHandle

If an invalid 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_INVALID_HANDLE (0xc0000008) 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 dw fs:[eax]
mov  fs:[eax], esp
;any illegal value will do
;must be dword-aligned on Vista
push esp
call CloseHandle
```

To defeat 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 transparently hooking the kernel32 AddVectoredExceptionHandler() function, in order to prevent another handler from registering as the first handler. However, it is still better than the problem of transparently hooking the ntdll NtClose() on *Windows NT* and *Windows 2000*, in order to register a Structured Exception Handler to hide the exception.

ix. OutputDebugString

The kernel32 OutputDebugString() function can demonstrate different behaviour, depending on whether or not a debugger is present. The most obvious difference in behaviour that the kernel32 GetLastError() function will return zero if a debugger is present.

Example code looks like this:

```
push 0
push esp
call OutputDebugStringA
call GetLastError
test eax, eax
je   being_debugged
```

x. ReadFile

The kernel32 ReadFile() function can be used as a technique for self-modification, by reading file content into the code stream. It is also an effective method for removing software breakpoints that a debugger might place. This is a technique that I discussed privately in 1999, but it was described publicly by Piotr Bania[vi] in 2007.

Example code looks like this:

```
xor  ebx, ebx
mov  ebp, offset l2
```

```
     push 104h ;MAX_PATH
     push ebp
     push ebx ;self filename
     call GetModuleFileNameA
     push ebx
     push ebx
     push 3 ;OPEN_EXISTING
     push ebx
     push 1 ;FILE_SHARE_READ
     push 80000000h ;GENERIC_READ
     push ebp
     call CreateFileA
     push ebx
     push esp
     ;more bytes might be more useful
     push 1
     push offset l1
     push eax
     call ReadFile
     ;replaced by "M"
     ;from the MZ header
l1:  int  3
     ...
l2:  db   104h dup (?);MAX_PATH
```

The way to defeat this technique is to use hardware breakpoints instead of software breakpoints after the API call.

xi. WriteProcessMemory

The kernel32 WriteProcessMemory() function technique is a simple variation on the kernel32 ReadFile() function technique above, but it requires that the data to write are already present in the process memory space.

Example code looks like this:

```
     push 1
     push offset l1
     push offset l2
     push -1 ;GetCurrentProcess()
     call WriteProcessMemory
l1:  nop
l2:  int  3
```

This technique is used by *NsAnti*. The way to defeat this technique is to use hardware breakpoints instead of software breakpoints after the API call.

xii. UnhandledExceptionFilter

When an exception occurs, and no registered Structured Exception Handlers (neither Safe nor Legacy) or Vectored Exception Handlers exist, or none of the registered handlers handles the exception, the kernel32 UnhandledExceptionFilter() function will be called as a last resort. Within that function is a call to the handler that was registered by the kernel32 SetUnhandledExceptionFilter() function, but that call will not reached if a debugger is present. Instead, the exception will be passed to the debugger. The presence of a debugger is determined by a call to the ntdll NtQueryInformationProcess (ProcessDebugPort class) function. So, for applications that do not know about the ntdll NtQueryInformationProcess (ProcessDebugPort class) function, the missing exception can be used to infer the presence of the debugger.

Example code looks like this:

```
     push offset l1
     call SetUnhandledExceptionFilter
     ;force an exception to occur
     int  3
     jmp  being_debugged
l1:  ...
```

xiii. Block Input

The user32 BlockInput() function blocks mouse and keyboard events from reaching applications. It is a very effective way to disable debuggers.

Example code looks like this:

```
     push 1
     call BlockInput
```

This technique is used by *Yoda's Protector*, among others.

xiv. SuspendThread

The kernel32 SuspendThread() function can be another very effective way to disable user-mode debuggers like *OllyDbg* and *Turbo Debug*. This can be achieved by enumerating the processes, as described above, then suspending the main thread of the parent process, if it does not match "Explorer.exe". This technique is used by *Yoda's Protector*.

xv. Guard Pages

Guard pages can be used for a simple debugger detection. An exception handler is registered, an executable/writable page is allocated dynamically, a "C3" opcode ("RET" instruction) is written to it, and then the page protection is changed to PAGE_GUARD. Then an attempt is made to execute the instruction. This should result in an EXCEPTION_GUARD_PAGE (0x80000001) exception being received by the exception handler, but if a debugger is present, the debugger might intercept the exception and allow the execution to continue. In fact, that's exactly what happens in *OllyDbg* (see Anti-debugging:*OllyDbg* section below).

Example code looks like this:

```
      xor  ebx, ebx
      push 40h ;PAGE_EXECUTE_READWRITE
      push 1000h ;MEM_COMMIT
      push 1
      push ebx
      call VirtualAlloc
      mov  b [eax], 0c3h
      push eax
      push esp
      ;PAGE_EXECUTE_READWRITE
      ;+ PAGE_GUARD
      push 140h
      push 1
      push eax
      xchg ebp, eax
      call VirtualProtect
      push offset l1
      push dw fs:[ebx]
      mov  fs:[ebx], esp
      push offset being_debugged
      ;executing ret will branch
      ;to being_debugged
      jmp  ebp
      ;an exception will reach here
  l1: ...
```

This technique is used by *PC Guard*.

xvi. Alternative desktop

Windows NT-based platforms support multiple desktops per session. It is possible to select a different active desktop, which has the effect of hiding the windows of the previously active desktop, and with no obvious way to switch back to the old desktop.

Example code looks like this:

```
      xor  eax, eax
      push eax
      ;DESKTOP_CREATEWINDOW
      ;+ DESKTOP_WRITEOBJECTS
      ;+ DESKTOP_SWITCHDESKTOP
      push 182h
      push eax
      push eax
      push eax
      push offset l1
      call CreateDesktopA
      push eax
      call SwitchDesktop
      ...
  l1: db   "mydesktop", 0
```

This technique is used by *HyperUnpackMe2*.

e. Hardware tricks

i. Prefetch queue

Given this code:

```
l1: call l3
l2: ...
l3: mov  al, 0c3h
    mov  edi, offset l3
    or   ecx, -1
    rep  stosb
```

What happens next? The answer depends on several things. Clearly, the code overwrites itself, which might lead one to conclude that it stops as soon as the REP is destroyed. If a debugger is used to single-step, then that is exactly what happens. However, if a debugger is not present, then the write continues until an exception occurs. Which exception that is depends on the memory layout at the time.

If, after the code, is a purely virtual region that has been accessed by a debugger, for example, then an access violation exception will occur and the program will exit if no exception handler has been registered.

On the other hand, if the virtual memory has not been accessed, then the REP will stop. No *visible* exception will occur, but a "C3" opcode ("RET" instruction) will be executed, and control will be returned to l2.

Why? The answer is the prefetch queue. On the x86 family of CPUs prior to the *Pentium*, the prefetch queue would not be flushed automatically when a memory write occurred at an address that corresponded to the address of the bytes in the prefetch queue. However, the queue would be flushed whenever an exception occurred, such as the single-step exception that many debuggers use to step through code. This behaviour allowed for all kinds of anti-debugger tricks, mostly concerned with overwriting the next instruction to execute. In the absence of a debugger, the prefetch queue would execute the original instruction. In the presence of a debugger that triggers a single-step exception, the queue would be flushed, and the alteration would be applied.

Intel considered this behaviour to be a bug, and it was fixed in the *Pentium* and later CPUs, but with two exceptions that remain to this day: the REP MOVS and REP STOS instructions. For those two instructions, the CPU still caches them and continues to execute them even when the instruction sequence has been overwritten in memory.

The execution continues until completion, or until an exception occurs. In the case above, an exception occurs, but it is a page fault when the value in EDI reaches a reserved page in memory. At that time, the CPU flushes and reloads the prefetch queue, sees the "C3" opcode ("RET" instruction) where the REP STOS instruction was previously, and executes that instead.

This technique is used by *Invius*.

Another example of this trick exists that does not rely on the page fault.
Example code looks like this:

```
l1: mov  al, 90h
    push 10h
    pop  ecx
    mov  edi, offset l1
    rep  stosb
    ...
```

One variation contains a JMP instruction within the altered range; the other contains a JECXZ instruction outside of the altered range. They have opposite effects.

In both cases, the "90" opcode ("NOP" instruction) in the AL register is used to overwrite the REP STOSB and some of the following bytes. Incorrect emulation (or single-stepping through the code, as with a debugger) will cause the REP to exit prematurely, allowing the instructions immediately following the STOSB instruction to execute. In the first variation, the JMP instruction will be executed as a result, revealing the presence of a debugger or similar. In the second variation, the value in ECX will be zero only if the REP STOSB completes. If the JECXZ instruction is not executed, this reveals the presence of a debugger or similar.

The JECXZ version of this technique is used by *Obsidium*.

The *Pentium Pro* introduced an additional behaviour, called a "fast string" operation, which is also supported by modern CPUs. It is available for both MOVS and STOS. It requires these conditions: REP prefix, EDI aligned to a multiple of 8 bytes (and ESI, too, for MOVS on a *Pentium* 3), ESI and EDI at least a cache-line apart (64 bytes for the *Pentium* 4 and later CPUs, 32 bytes for earlier CPUs) for MOVS, ECX at least 64, D flag clear in the EFLAGS register, and WB or WC memory type for EDI (and ESI for MOVS). Additionally, a Model Specific Register (MSR) must be set appropriately (though by default it is already enabled) - either 1A0 bit 0 or 1E0 bit 2. Of particular interest is that the single-step exception *cannot* interrupt the operation.

ii. Hardware Breakpoints

When an exception occurs, *Windows* passes to the exception handler a context structure which contains the values of the general registers, segment registers, control registers, and the debug registers. If a debugger is present and passes the exception to the debuggee with hardware breakpoints in use, then the debug registers will contain values that reveal the presence of the debugger.
Example code looks like this:

```
    xor  eax, eax
    push offset l1
    push dw fs:[eax]
    mov  fs:[eax], esp
    ;force an exception to occur
    jmp  eax
    ...
    ;ContextRecord
l1: mov  eax, [esp+0ch]
    mov  eax, [eax+4] ;Dr0
    or   eax, [eax+8] ;Dr1
    or   eax, [eax+0ch] ;Dr2
    or   eax, [eax+10h] ;Dr3
    jne  being_debugged
```

The debugger is also vulnerable to being bypassed if the debuggee erases the contents of the debug registers prior to resuming execution after the exception. This technique is used by *ASProtect*, among others.

iii. Instruction Counting

Instruction counting can be performed by registering an exception handler, then setting some hardware breakpoints on particular addresses. When each address is hit, an EXCEPTION_SINGLE_STEP (0x80000004) exception will be raised. This exception will be passed to the exception handler, which can adjust the instruction pointer to point to a new instruction, and then resume execution. To set the breakpoints requires access to a context structure. This can be achieved by calling the kernel32 GetThreadContext() function. Alternatively, a context structure is passed to an exception handler, so by forcing an exception to occur, the context can be acquired in a more obfuscated manner. Some debuggers do not handle correctly hardware breakpoints that they did not set themselves, leading to some instructions not being counted by the exception handler.

Example code looks like this:

```
    xor  eax, eax
    cdq
    push offset l5
    push dw fs:[eax]
    mov  fs:[eax], esp
    int  3
l1: nop
l2: nop
l3: nop
l4: nop
    div  edx
    cmp  al, 4
    jne  being_debugged
    ...
l5: xor  eax, eax
    ;ExceptionRecord
    mov  ecx, [esp+4]
    ;ContextRecord
    mov  edx, [esp+0ch]
    ;CONTEXT_Eip
    inc  b [edx+0b8h]
    ;ExceptionCode
    mov  ecx, [ecx]
    ;EXCEPTION_INT_DIVIDE_BY_ZERO
    cmp  ecx, 0c0000094h
    jne  l6
    ;CONTEXT_Eip
    inc  b [edx+0b8h]
    mov  [edx+4], eax ;Dr0
    mov  [edx+8], eax ;Dr1
    mov  [edx+0ch], eax ;Dr2
    mov  [edx+10h], eax ;Dr3
    mov  [edx+14h], eax ;Dr6
    mov  [edx+18h], eax ;Dr7
    ret
    ;EXCEPTION_BREAKPOINT
l6: cmp  ecx, 80000003h
    jne  l7
    ;Dr0
    mov  dw [edx+4], offset l1
    ;Dr1
    mov  dw [edx+8], offset l2
    ;Dr2
    mov  dw [edx+0ch], offset l3
    ;Dr3
    mov  dw [edx+10h], offset l4
    ;Dr7
    mov  dw [edx+18h], 155h
    ret
    ;EXCEPTION_SINGLE_STEP
l7: cmp  ecx, 80000004h
    jne  being_debugged
    ;CONTEXT_Eax
    inc  b [edx+0b0h]
    ret
```

This technique is used by *tELock*.

iv. Execution Timing

When a debugger is present, and used to single-step through the code, there is a significant delay between the executions of the individual instructions, when compared to native execution. This delay can be measured using one of several possible time sources. These sources include the RDTSC instruction, the kernel32 GetTickCount() function, and the winmm timeGetTime() function, among others. However, the resolution of the winmm timeGetTime() function is variable, depending on whether or not it branches internally to the kernel32 GetTickCount() function, making it very unreliable to measure small intervals.

Example code looks like this for RDTSC:

```
rdtsc
xchg  ecx, eax
rdtsc
sub   eax, ecx
cmp   eax, 500h
jnbe  being_debugged
```

Example code looks like this for kernel32 GetTickCount():

```
call GetTickCount
xchg ebx, eax
call GetTickCount
sub  eax, ebx
cmp  eax, 1
jnb  being_debugged
```

Example code looks like this for winmm timeGetTime():

```
call timeGetTime
xchg ebx, eax
call timeGetTime
sub  eax, ebx
cmp  eax, 10h
jnb  being_debugged
```

v. EIP via Exceptions

Using exceptions to alter the value of eip is a very common technique among packers. It serves as an effective anti-debugging technique, since debuggers typically intercept some of the exceptions (int 1 and int 3, for example). It also provides for a level of obfuscation, particularly if the exception trigger is not immediately obvious.

Example code looks like this:

```
xor  eax, eax
push offset l3
```

```
     push dw fs:[eax]
     mov  fs:[eax], esp
 l1: call l1
 l2: jmp  l2
 l3: pop  eax
     pop  eax
     pop  esp
 l4: ...
```

Is l2 ever reached? No, it's not. A stack overflow exception occurs at l1, causing a transfer of control to l3. After the stack is restored, execution continues from l4. This technique is used by *PECompact*, among others.

f. Process tricks

i. Header entrypoint

Any section of the file, whose attributes do not include IMAGE_SCN_MEM_WRITE (writable) and/or IMAGE_SCN_MEM_EXECUTE (executable), is read-only by default to a remote debugger. This includes the PE header, since there is no section that describes it (there is an exception to this, see Anti-Emulating:File-Format section below). If the entrypoint happens to be in such a section, then a debugger will not be able to successfully set any breakpoints, if it does not first call the kernel32 VirtualProtectEx() function to write-enable the memory region. Further, if the failure to set the breakpoint is not noticed by the debugger, then the debugger might allow the debuggee to run freely. This is the case for *Turbo Debugger*. This technique is used by *MEW*, among others.

ii. Parent process

Users usually execute applications manually via a window provided by the shell. As a result, the parent of any such process will be Explorer.exe. Of course, if the application is executed from the command-line, then the command-line application will be the parent. Executing applications from the command-line can be a problem for certain packers. This is because some packers check the parent process name, expecting it to be "Explorer.exe", or the packers compare the parent process ID against that of Explorer.exe. A mismatch in either case is then assumed to be caused by a debugger creating the process.

The process ID of both Explorer.exe, and the parent of the current process, can be obtained by the kernel32 CreateToolhelp32Snapshot() function and a kernel32 Process32Next() function enumeration.

Example code looks like this:

```
     xor   esi, esi
     xor   edi, edi
     push  esi
     push  2 ;TH32CS_SNAPPROCESS
     call  CreateToolhelp32Snapshot
     mov   ebx, offset l5
     push  ebx
     push  eax
     xchg  ebp, eax
     call  Process32First
 l1: call  GetCurrentProcessId
     ;th32ProcessID
     cmp   [ebx+8], eax
     ;th32ParentProcessID
     cmove edi, [ebx+18h]
     test  esi, esi
     je    l2
     test  edi, edi
     je    l2
     cmp   esi, edi
     jne   being_debugged
 l2: lea   ecx, [ebx+24h] ;szExeFile
     push  esi
     mov   esi, ecx
 l3: lodsb
     cmp   al, "\"
     cmove ecx, esi
     or    b [esi-1], " "
     test  al, al
     jne   l3
     sub   esi, ecx
     xchg  ecx, esi
     push  edi
     mov   edi, offset l4
     repe  cmpsb
     pop   edi
     pop   esi
     ;th32ProcessID
     cmove esi, [ebx+8]
     push  ebx
     push  ebp
     call  Process32Next
     test  eax, eax
     jne   l1
     ...
 l4: db    "explorer.exe "
     ;sizeof(PROCESSENTRY32)
 l5: dd    128h
     db    124h dup (?)
```

This technique is used by *Yoda's Protector*, among others. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger. However, a common technique is to force the kernel32 Process32Next() function to return FALSE, which causes the loop to exit early. It should be a suspicious condition if either Explorer.exe or the current processes were not seen, but *Yoda's Protector* (and some other packers) does not contain any requirement that both were found.

The process ID of both Explorer.exe, and the parent of the current process, can be obtained by the ntdll NtQuerySystemInformation (SystemProcessInformation (5)) function.

Example code looks like this:

```
    xor   ebp, ebp
    xor   esi, esi
    xor   edi, edi
    jmp   l2
l1: push  8000h ;MEM_RELEASE
    push  esi
    push  ebx
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   ebp, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    push  eax
    push  ebp
    push  esi
    call  VirtualAlloc
    ;function does not return
    ;required length for this class
    push  esi
    ;must calculate by brute-force
    push  ebp
    push  eax
    ;SystemProcessInformation
    push  5
    xchg  ebx, eax
    call  NtQuerySystemInformation
    ;STATUS_INFO_LENGTH_MISMATCH
    cmp   eax, 0c0000004h
    je    l1
l3: call  GetCurrentProcessId
    ;UniqueProcessId
    cmp   [ebx+44h], eax
    ;InheritedFromUniqueProcessId
    cmove edi, [ebx+48h]
    test  esi, esi
    je    l4
    test  edi, edi
    je    l4
    cmp   esi, edi
    jne   being_debugged
l4: mov   ecx, [ebx+3ch] ;ImageName
    jecxz l6
    push  esi
    xor   eax, eax
    mov   esi, ecx
l5: lodsw
    cmp   eax, "\"
    cmove ecx, esi
    push  ecx
    push  eax
    call  CharLowerW
    mov   w [esi-2], ax
    pop   ecx
    test  eax, eax
    jne   l5
    sub   esi, ecx
    xchg  ecx, esi
    push  edi
    mov   edi, offset l7
    repe  cmpsb
    pop   edi
    pop   esi
    ;UniqueProcessId
    cmove esi, [ebx+44h]
    ;NextEntryOffset
l6: mov   ecx, [ebx]
    add   ebx, ecx
    inc   ecx
    loop  l3
    ...
l7: dw "e","x","p","l","o","r"
    dw "e","r",".","e","x","e",0
```

However, the process ID of Explorer.exe can be obtained most simply by the user32 GetShellWindow() and user32 GetWindowThreadProcessId() functions. The process ID of the parent of the current process can be obtained most simply by the ntdll NtQueryInformationProcess (ProcessBasicInformation (0)) function.

Example code looks like this:

```
    call GetShellWindow
    push eax
    push esp
    push eax
    call GetWindowThreadProcessId
    push 0
  ;sizeof(PROCESS_BASIC_INFORMATION)
    push 18h
    mov  ebp, offset l1
    push ebp
    push 0 ;ProcessBasicInformation
    push -1 ;GetCurrentProcess()
    call NtQueryInformationProcess
    pop  eax
    ;InheritedFromUniqueProcessId
    cmp  [ebp+14h], eax
    jne  being_debugged
    ...
;sizeof(PROCESS_BASIC_INFORMATION)
l1: db   18h dup (?)
```

iii. Self-execution

One of the simplest ways to escape from the control of a debugger is for a process to execute another copy of itself. Typically, the process will use a synchronisation object, such as a mutex, to prevent infinite executions. The first process will create the mutex, and then execute the copy of the process. The second process will not be debugged, even if the first process was. It will also know that it is the copy since the mutex will exist.

Example code looks like this:

```
      xor  ebx, ebx
      push offset l2
      push eax
      push eax
      call CreateMutexA
      call GetLastError
      ;ERROR_ALREADY_EXISTS
      cmp  eax, 0b7h
      je   l1
      mov  ebp, offset l3
      push ebp
      call GetStartupInfoA
      call GetCommandLineA
      ;sizeof(PROCESS_INFORMATION)
      sub  esp, 10h
      push esp
      push ebp
      push ebx
      push ebx
      push ebx
      push ebx
      push ebx
      push ebx
      push eax
      push ebx
      call CreateProcessA
      pop  eax
      push -1 ;INFINITE
      push eax
      call WaitForSingleObject
      call ExitProcess
  l1: ...
  l2: db   "my mutex", 0
      ;sizeof(STARTUPINFO)
  l3: db   44h dup (?)
```

A common mistake is the use of the kernel32 Sleep() function, instead of the kernel32 WaitForSingleObject() function, because it introduces a race condition. The problem occurs when there is CPU-intensive activity. This could be because of a sufficiently complicated protection (or intentional delays) in the second process; but also actions that the user might perform while the execution is in progress, such as browsing the network or extracting files from an archive. The result is that the second process might not reach the mutex check before the delay expires; leading it to think that it is the first process. The result is that it executes yet another copy of the process. This behaviour can be repeated any number of times, until one of the processes completes the mutex check successfully. This technique is used by *MSLRH*, and the exact problem is present there.

iv. Process Name

As noted above, the list of process names can be retrieved by the kernel32 CreateTool32Snapshot() function, or the ntdll QuerySystemInformation() function. In addition to finding Explorer.exe or the current process name, some packers look for other process names, particularly those which belong to anti-malware vendors or specialised tools.

Example code looks like this for kernel32 CreateToolhelp32Snapshot():

```
      push  0
      push  2 ;TH32CS_SNAPPROCESS
      call  CreateToolhelp32Snapshot
      mov   ebx, offset l5
      push  ebx
      push  eax
      xchg  ebp, eax
      call  Process32First
  l1: lea   ecx, [ebx+24h] ;szExeFile
      mov   esi, ecx
  l2: lodsb
      cmp   al, "\"
      cmove ecx, esi
      or    b [esi-1], " "
      test  al, al
      jne   l2
      sub   esi, ecx
      xchg  ecx, esi
      mov   edi, offset l4
  l3: push  ecx
      push  esi
      repe  cmpsb
      je    being_debugged
      mov   al, " "
      not   ecx
      ;move to previous character
      dec   edi
      ;then find end of string
      repne scasb
      pop   esi
      pop   ecx
      cmp   [edi], al
      jne   l3
      push  ebx
      push  ebp
      call  Process32Next
      test  eax, eax
      jne   l1
      ...
  l4: <array of space-terminated ASCII
strings, space to end>
      ;sizeof(PROCESSENTRY32)
  l5: dd   128h
      db   124h dup (?)
```

Example code looks like this for ntdll NtQuerySystemInformation():

```
    xor   ebp, ebp
    xor   esi, esi
    jmp   l2
l1: push  8000h ;MEM_RELEASE
    push  esi
    push  ebx
    call  VirtualFree
l2: xor   eax, eax
    mov   ah, 10h ;MEM_COMMIT
    add   ebp, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    push  eax
    push  ebp
    push  esi
    call  VirtualAlloc
    ;function does not return
    ;required length for this class
    push  esi
    ;must calculate by brute-force
    push  ebp
    push  eax
    ;SystemProcessInformation
    push  5
    xchg  ebx, eax
    call  NtQuerySystemInformation
    ;STATUS_INFO_LENGTH_MISMATCH
    cmp   eax, 0c0000004h
    je    l1
l3: mov   ecx, [ebx+3ch] ;ImageName
    jecxz l6
    xor   eax, eax
    mov   esi, ecx
l4: lodsw
    cmp   eax, "\"
    cmove ecx, esi
    push  ecx
    push  eax
    call  CharLowerW
    mov   w [esi-2], ax
    pop   ecx
    test  eax, eax
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    mov   edi, offset l7
l5: push  ecx
    push  esi
    repe  cmpsb
    je    being_debugged
    not   ecx
    ;move to previous character
    dec   edi
    ;force word-alignment
    and   edi, -2
    ;then find end of string
    repne scasw
    pop   esi
    pop   ecx
    cmp   [edi], ax
    jne   l5
    ;NextEntryOffset
l6: mov   ecx, [ebx]
    add   ebx, ecx
    inc   ecx
    loop  l3
    ...
    ;must be word-aligned
    ;for correct scanning
    align 2
l7: <array of null-terminated
Unicode strings, null to end>
```

v. Threads

Threads are used by some packers to perform actions such as periodically checking for the presence of a debugger, or ensuring the integrity of the main code. The use of threads had an additional advantage early on, which was that some anti-malware emulators did not support threads, allowing the packed file to cause an early exit.

Example code looks like this:

```
l1: xor   eax, eax
    push  eax
    push  esp
    push  eax
    push  eax
    push  offset l2
    push  eax
    push  eax
    call  CreateThread
    ...
l2: xor   eax, eax
    cdq
    mov   ecx, offset l4 - offset l1
    mov   esi, offset l1
l3: lodsb
    ;simple sum
    ;to detect breakpoints
    add   edx, eax
    loop  l3
    cmp   edx, <checksum>
    jne   being_debugged
    ;small delay then restart
    push  100h
    call  Sleep
    jmp   l2
l4: ;code end
```

This technique is used by *PE-Crypt32*, among others.

vi. Self-debugging

Self-debugging is the act of running a copy of a process, and attaching to it as a debugger. Since only one debugger can be attached to a process at any point in time, the copy of the process becomes undebuggable by ordinary means.

Example code looks like this:

```
    xor  ebx, ebx
    mov  ebp, offset l3
    push ebp
    call GetStartupInfoA
    call GetCommandLineA
    mov  esi, offset l4
    push esi
    push ebp
    push ebx
    push ebx
    push 1 ;DEBUG_PROCESS
    push ebx
    push ebx
    push ebx
    push eax
    push ebx
    call CreateProcessA
    mov  ebx, offset l5
    jmp  l2
l1: push 10002h ;DBG_CONTINUE
    push dw [esi+0ch] ;dwThreadId
    push dw [esi+8] ;dwProcessId
    call ContinueDebugEvent
l2: push -1 ;INFINITE
    push ebx
    call WaitForDebugEvent
    cmp  b [ebx], 5
;EXIT_PROCESS_DEBUG_EVENT
    jne  l1
    ...
    ;sizeof(STARTUPINFO)
l3: db   44h dup (?)
    ;sizeof(PROCESS_INFORMATION)
l4: db   10h dup (?)
    ;sizeof(DEBUG_EVENT)
l5: db   60h dup (?)
```

This technique is used by *Armadillo*, among others. This technique can be defeated most easily by kernel-mode code zeroing the EPROCESS->DebugPort field. Doing so will allow another debugger to attach to the process. The debugged process can also be opened via the kernel32 OpenProcess() function, which means that a DLL can be injected into the process space. Alternatively, on *Windows XP* and later, the kernel32 DebugActiveProcessStop() function can be used to detach the debugger.

vii. Disassembly

Some packers examine not just the first few bytes of an API for breakpoints, but actively disassemble the function code. There are a few reasons why they might do that. One reason is in order to perform API interception, whereby some complete instructions from the function are copied to a private buffer and executed from there. A jump is placed at the end of those instructions, to point after the last copied instruction in the original API code. This has the effect of bypassing breakpoints that are placed anywhere within the first few instructions of the original API code.

Another reason is in order to perform a more reliable search for breakpoints. By knowing the location of the instruction boundaries, there is no risk of encountering what appears to be a breakpoint, but is actually some data. For example, 0xb8 0xcc 0x00 0x00 0x00 appears to contain a breakpoint, but when disassembled and displayed, the sequence is "MOV EAX, 000000CC".

In addition to searching for breakpoints, some packers search for detours. Detours are jump instructions that are inserted, usually as the first instruction, to point to a private location. The code at that private location typically creates a log of the APIs that are called, though it is not restricted to that behaviour. The problem with detecting detours is that it also detects hot-patching. *Microsoft* added a dummy instruction to many functions in *Windows XP*, that allows a jump instruction to be placed cleanly (that is, without concern for instruction boundaries, since the dummy instruction achieves the required alignment). This jump instruction would point to a private location that contains code to deal with a vulnerability in the hooked function. If a packer detects detours and refuses to run if a detour of any kind is found, then it will also refuse to run if a function has been hot-patched.

viii. TLS Callback

This is a technique that allows the execution of user-defined code before the execution of the main entrypoint code. It is a technique that I discussed privately in 2000, but it was demonstrated publicly by Radim Picha[vii] later that same year. It was used in a virus[viii] in 2002. It has been used by *ExeCryptor* and others since 2004.

ix. Device names

Tools that make use of kernel-mode drivers also need a way to communicate with those drivers. A very common method is through the use of named devices. Thus, by attempting to open such a device, any success 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>
```

A typical list includes the following names:

\\.\SICE
\\.\SIWVID
\\.\NTICE

These names belong to *SoftICE*. Note that a successful opening of the device does not mean that *SoftICE* is active, but that it is present. However, that is sufficient for many people. The first two drivers are present on *Windows 9x*-based platforms, the third driver is present on *Windows NT*-based platforms, but a lot of copy/paste occurs in the packer space, so this list appears often, even in packers that do not run on *Windows 9x*-based platforms.

Other common device names include these:

\\.\REGVXG
\\.\REGSYS

These names belong to RegMon. The first name is for *Windows 9x*-based platforms, the second name is for *Windows NT*-based platforms.

\\.\FILEVXG
\\.\FILEM

These names belong to FileMon. The first name is for *Windows 9x*-based platforms, the second name is for *Windows NT*-based platforms.

\\.\TRW

This name belongs to TRW. TRW is a debugger for only *Windows 9x*-based platforms, yet some packers check for it even on *Windows NT*-based platforms.

\\.\ICEEXT

This name belongs to *SoftICE* extender.

g. *SoftICE*-specific

For many years, *SoftICE* was the most popular of debuggers for the *Windows* platform. It is a debugger that makes use of a kernel-mode driver, in order to support debugging of both user-mode and kernel-mode code, including transitions in either direction between the two.

SoftICE contains a number of vulnerabilities. A description of them is beyond the scope of this paper. A companion paper (Anti-Unpacking Tricks - Future) will cover the topic in detail.

i. Driver information

The names of the device drivers on the system can be enumerated. This can be achieved using the ntdll NtQuerySystemInformation (SystemModuleInformation (0x0b)) function. For each module that is returned, the version information in the file can be retrieved using the version VerQueryValue() function. This information typically includes the Product Name and Copyright strings, which can be matched against specific products and companies, such as "*SoftICE*", "*Compuware*", and "*NuMega*".

ii. Interrupt 1

The interrupt 1 descriptor normally has a descriptor privilege level (DPL) of 0, which means that the "cd 01" opcode ("int 1" instruction) cannot be issued from ring 3. An attempt to execute this interrupt directly will result in a general protection fault ("int 0x0d" exception) being issued by the CPU, eventually resulting in an EXCEPTION_ACCESS_VIOLATION (0xc0000005) exception being raised by *Windows*.

However, if *SoftICE* is running, it hooks interrupt 1 and adjusts the DPL to 3, so that *SoftICE* can single-step through user-mode code. This is not visible from within *SoftICE*, though - the "IDT" command, to display the interrupt descriptor table, shows the original interrupt 1 handler address with a DPL of 0, as though *SoftICE* were not present.

The problem is that when an interrupt 1 occurs, *SoftICE* does not check if it was caused by the trap flag or by a software interrupt. The result is that *SoftICE* always calls the original interrupt 1 handler, and an EXCEPTION_SINGLE_STEP (0x80000004) exception is raised instead of the

EXCEPTION_ACCESS_VIOLATION (0xc0000005) exception, allowing for an easy detection method.

Example code looks like this:

```
     xor  eax, eax
     push offset l1
     push dw fs:[eax]
     mov  fs:[eax], esp
     int  1
     ...
     ;ExceptionRecord
l1:  mov  eax, [esp+4]
     ;EXCEPTION_SINGLE_STEP
     cmp  dw [eax], 80000004h
     je   being_debugged
```

This technique is used by *SafeDisc*. To defeat this technique might appear to be a simple matter of restoring the DPL of interrupt 1. It is not so simple. The problem is to determine reliably the cause of an exception at the interrupt 0x0d level. The instruction queue can be examined for an "int 1" sequence, but the trap flag could also appear to be set at the same time, even though it did not become active. This can happen if interrupts are delayed for one instruction (via "pop ss", for example), then the trap flag will not be responsible for the exception, even though it is set. A companion paper (Anti-Unpacking Tricks - Future) will cover some additional aspects of this problem.

h. *OllyDbg*-specific

OllyDbg is perhaps the most popular of user-mode debuggers. It supports plug-ins. Some packers have been written to detect *OllyDbg*, so some plug-ins have been written to attempt to hide *OllyDbg* from those packers. Correspondingly, other packers have been written to detect these plug-ins. A description of those plug-ins, and the vulnerabilities in them, is beyond the scope of this paper. A companion paper (Anti-Unpacking Tricks - Future) will cover the topic in detail.

i. Malformed files

OllyDbg is too strict regarding the Portable Executable format - it will refuse to open a file whose data directories do not end at exactly the end of the Optional Header. It attempts to allocate the amount of memory specified by the Export Directory Size, Base Relocation Directory Size, Export Address Table Entries, and PE->SizeOfCode fields, regardless of how large the values are. This can cause the operating system swap file to grow enormously, which has a significant performance impact on the system.

ii. Initial esi value

The esi register has an initial value of 0xffffffff in *OllyDbg* on *Windows XP*, which seems to be constant, leading some people to use it as a detection method[ix]. In fact, it's just a coincidence (and the initial value is 0 on *Windows 2000*). The value is a remnant of an exception handler structure that *Windows XP* created during a call to the ntdll RtlAllocateHeap() function. That location of that value corresponds to the esi member in the context structure that is created by the kernel32 CreateProcess() function. The kernel32 CreateProcess() function does not initialise the esi member.

iii. OutputDebugString

OllyDbg passes user-defined data directly to the msvcrt _vsprintf() function. If those data contain formatting string tokens, particularly if multiple "%s" tokens are used, then it is likely that one of them will point to an invalid memory region and crash *OllyDbg*.

iv. FindWindow

OllyDbg can be found by calling the user32 FindWindow() function, and passing "OLLYDBG" as the class name to find.

Example code looks like this:

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

v. Guard Pages

OllyDbg uses guard pages to handle memory breakpoints. As noted above, if an application places executable instructions in a guarded page, an attempt to execute them should result in an exception, but in *OllyDbg* they will be executed instead.

i. *HideDebugger*-specific

HideDebugger is a plug-in for *OllyDbg*. Early versions of *HideDebugger* hooked the debuggee's kernel32 OpenProcess() function. The hook was done by placing a far jump to a new handler, at offset 6 in the kernel32 OpenProcess() function. The presence of the jump was a good indicator that the *HideDebugger* plug-in was present.

Example code looks like this:

```
     push offset l1
     call GetModuleHandleA
     push offset l2
     push eax
     call GetProcAddress
     cmp  b [eax+6], 0eah
     je   being_debugged
     ...
l1:  db   "kernel32", 0
l2:  db   "OpenProcess", 0
```

j. *ImmunityDebugger*-specific

ImmunityDebugger is essentially *OllyDbg* with a Python command-line interface. In fact, it is largely byte-for-byte identical to the *OllyDbg* code. Correspondingly, it has the same vulnerabilities as *OllyDbg*, with respect to both detect and exploitation.

k. *WinDbg*-specific

i. FindWindow

WinDbg can be found by calling the user32 FindWindow() function, and passing "WinDbgFrameClass" as the class name to find.

Example code looks like this:

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

l. Miscellaneous tools

i. FindWindow

There are several less common tools that are of interest to some packers, such as window name of "Import REConstructor v1.6 FINAL (C) 2001-2003 MackT/uCF", or a class name of "TESTDBG", "kk1, "Eew57", or "Shadow". These names are checked by *MSLRH*.

III. ANTI-UNPACKING BY ANTI-EMULATING

Some methods to detect emulators and virtual machines have been described elsewhere[x]. Some additional methods are described here. A companion paper (Anti-Unpacking Tricks - Future) will describe some further methods.

a. Software Interrupts

i. Interrupt 3

When an EXCEPTION_BREAKPOINT (0x80000003) occurs, the eip register has already been advanced to the next instruction, so *Windows* wants to rewind the eip to point to the proper place. The problem is that *Windows* assumes that the exception is caused by a single-byte "CC" opcode (short form "INT 3" instruction). If the "CD 03" opcode (long form "INT 3" instruction) is used to cause the exception, then the eip will be pointing to the wrong location. The same behaviour can be seen if any prefixes are placed before the short-form "INT 3" instruction. An emulator that does not behave in the same way will be revealed instantly. This technique is used by *TryGames*.

b. Time-locks

Time-locks are a very effective anti-emulation technique. Most anti-malware emulators intentionally contain a limit to the amount of time and/or the number of CPU instructions that can be emulated, before the emulator will exit with no detection. This behavior is almost a requirement, since a user will typically not be patient enough to wait for an emulated application to exit on its own (if it ever would), before being able to access it normally. This leads to a vulnerability, whereby an attacker will produce a sample which intentionally delays its main execution, usually via a dummy loop, in an attempt to force an emulator to give up.

Example code looks like this:

```
     mov  ecx, 400000h
l1:  loop l1
```

In some cases, such dummy loops can be recognized and skipped, but in that case, care must be taken to adjust the values of any internal timers, and also the CPU registers that are involved. Otherwise, the arbitrary skipping of the loop might be detected.

Example code looks like this:

```
     call GetTickCount
     xchg ebx, eax
     mov  ecx, 400000h
l1:  loop l1
     call GetTickCount
     sub  eax, ebx
     cmp  eax, 1000h
     jbe  being_debugged
```

Further, the loop might not be a dummy one at all, in the sense that the results might be used for a real purpose, even though they could have been calculated without resorting to a loop.

Real-world example code looks like this:

```
     mov ebp, esp
     mov ebp, [ebp+1ch] ;0ffffffffh
     sub ebp, 5
l1:  sub ebp, 0ah
     dec eax
     or  ebp, ebp
     jne l1
```

In this case, the calculated value is also used as a key, so the loop cannot be skipped arbitrarily. This technique is used by *Tibs*.

c. Invalid API parameters

Many APIs return error codes when they receive invalid parameters. The problem for anti-malware emulators is that, for simplicity, such error checking is not implemented. This leads to a vulnerability, whereby an attacker will intentionally pass known invalid parameters to the function, and expecting an error code to be returned. In some cases, this error code is used as a key for decryption. Any emulator that fails to return the error code will not be able to decrypt the data.

Example code looks like this:

```
     push 1
     push 1
     call Beep
     call GetLastError
     ;ERROR_INVALID_PARAMETER (0x57)
     push 5 ;sizeof(l2)
     pop  ecx
     xchg edx, eax
     mov  esi, offset l2
     mov  edi, esi
l1:  lodsb
     xor  al, dl
     stosb
     loop l1
     ...
l2:  db   3fh, 32h, 3bh, 3bh, 38h
;secret message
```

This technique is used by *Tibs*.

d. GetProcAddress

The kernel32 GetProcAddress() function is intended to return the address of a function exported by the specified module. Since there is a potentially unlimited number of possible functions which can be retrieved from an infinite number of modules, it is impossible for them all to be available in an emulated environment that is provided by an anti-malware emulator. However, even some expected functions might be missing from such an environment, because of their lack of likely requirement, such as the kernel32 GetTapeParameters() function. The problem is that some packers will exit early if not all function addresses could be retrieved. To defeat that, some anti-malware emulators will always return a value for the kernel32 GetProcAddress(), regardless of the parameters that are passed in. This leads to a vulnerability, whereby an attacker will intentionally pass known invalid parameters to the function, and expecting no function address to be returned. Any emulator that returns an address in such a situation will be revealed.

Example code looks like this:

```
     push offset l1
     push 12345678h ;illegal value
     call GetProcAddress
     test eax, eax
     jne  being_debugged
     ...
l1:  db   "myfunction", 0
```

This technique is used by *NsAnti*. It is a specific case of the general bad API problem from above.

e. GetProcAddress(internal)

Some anti-malware emulators export special APIs, which can be used to communicate with the host environment, for example. This technique has been published elsewhere[xi].

Example code looks like this:

```
     push offset l1
     call GetModuleHandleA
     push offset l2
     push eax
     call GetProcAddress
     test eax, eax
     jne  being_debugged
     ...
l1:  db   "kernel32", 0
l2:  db   "Aaaaaa", 0
```

f. "Modern" CPU instructions

Different CPU emulators have different capabilities. The problem for anti-malware emulators is that, for simplicity, some (in some cases, many) CPU instructions are not supported. This can include entire classes, such as FPU, *MMX*, and *SSE*, as well as less common instructions such as CMPXCHG8B. In addition, some instructions have slightly unexpected behaviours which might also not be supported, such as that the CMPXCHG instruction always writes to memory, regardless of the result. Some of these behaviours have attributes (particularly the CPU flags) that are marked as "undefined", but nothing is undefined in hardware. The challenge is to determine the algorithm to reproduce it.

Some packers use FPU and *MMX* instructions as do-nothing instructions, but the side-effect is that the anti-malware emulator might give up and fail to detect anything.

g. Undocumented instructions

Some packers make use of undocumented CPU instructions, for the same reason as they do for the modern CPU instructions. That is, an anti-malware emulator is less likely to support undocumented instructions or undocumented encodings of documented instructions, so it might give up and fail to detect anything. A list of these has been published elsewhere[xii].

h. Selector verification

Selector verification is used to ensure that the descriptor table layout matches the operating system platform, as returned by the kernel32 GetVersion() function, for example. On *Windows 9x*-based platforms, the value of the cs selector can exceed 0xff, but on *Windows NT*-based platforms, the value is always 0x1b for ring 3 code.

Example code looks like this:

```
    call   GetVersion
    test   eax, eax
    ;Windows 9x-based platform
    js     l1
    mov    eax, cs
    xor    al, al
    test   eax, eax
    jne    being_emulated
l1: ...
```

This technique is used by *MSLRH*, among others.

i. Memory layout

There are certain in-memory structures that are always in a predictable location. One of those is the RTL_USER_PROCESS_PARAMETERS, which appears at memory location 0x20000 in normal circumstances. Within that structure, the "DllPath" field exists at 0x20498, and the command-line at 0x205f8. This structure can be moved if PE->ImageBase value is 0x20000 or less. The reason for this is because the PE sections are mapped into memory first, then the environment (at 0x10000 by default, and occupying 64kb of virtual memory because of the behaviour of the memory allocation function that is used), then the process parameters. By accessing these fields directly, certain APIs, such as the kernel32 GetCommandLine() function, do not need to be called. This can make it difficult to know from where certain information is gathered, and anti-malware emulators might not include these structures at all. This technique is used by *TryGames*.

j. File-format tricks

There are many known file-format tricks, yet occasionally a new one will appear. This can be a significant problem for anti-malware emulators, since if the emulator is responsible for parsing the file-format, then incompatibilities can appear because of differences in the emulated operating system. For example, *Windows 9x*-based platforms use a hard-coded value for the size of the Optional Header, and ignore the PE->SizeOfOptionalHeader field. They also allow gaps in the virtual memory described by the section table. *Windows NT*-based platforms honour the value in the PE->SizeOfOptionalHeader field, and do not allow any gaps.

Typical tricks include:

i. Non-aligned SizeOfImage

The file-format documentation states that the value in the PE->SizeOfImage field should be a multiple of the value in the PE->SectionAlignment field, but this is not a requirement. Instead, *Windows* will round up the value as required.

ii. Overlapping structures

By adjusting the values of certain fields, it is possible to produce structures that overlap each other. The common targets are the MZ->lfanew field, to produce a PE header that appears inside the MZ header; the PE->SizeOfOptionalHeader field, to produce a section table that appears inside the DataDirectory array; and the Import Address Table and Import Lookup Table virtual addresses, to produce an import table which has fields inside the PE header.

iii. Non-standard NumberOfRvaAndSizes

A common mistake is to assume that the value in the PE->NumberOfRvaAndSizes field is set to the value that exactly fills the Optional Header, and that the section table follows immediately. The proper method to calculate the location of the section table is to use the PE->SizeOfOptionalHeader field. Both *SoftICE* and *OllyDbg* contain this mistake. A companion paper (Anti-Unpacking Tricks - Future) will cover the implications in detail.

iv. Non-aligned SizeOfRawData

The SizeOfRawData field in the section table is another field that is subject to automatic rounding up by *Windows*. By relying on this behavior, it is possible to produce a section whose entrypoint appears to reside in purely virtual memory, but because of rounding, will have physical data to execute.

v. Non-aligned PointerToRawData

The PointerToRawData field in the section table is a field that is subject to automatic rounding down by *Windows*. By relying on this behaviour, it is possible to produce a section whose entrypoint appears to point to data other than what will actually be executed.

vi. No section table

An interesting thing happens if the value in the PE->SectionAlignment field is reduced to less than 4kb. Normally, the section that contains the PE header is neither writable nor executable, since there is no section table entry that describes it. However, if the value in the PE->SectionAlignment field is less than 4kb, then the PE header is marked internally as both writable and executable. Further, the contents of the section table become *optional*. That is, the entire section table can be zeroed out, and the file will be mapped as though it were one section whose size is equal to the value in the PE->SizeOfImage field.

IV. ANTI-UNPACKING BY ANTI-INTERCEPTING

a. Write->Exec

Some unpacking tools work by intercepting the execution of newly written pages, to guess when the unpacker has completed its main function and transferred control to the host. By writing then executing a dummy instruction, an unpacker can cause an intercepter to exit early.

Example code looks like this:

```
    mov [offset dest], 0c3h
    call dest
```

This technique is used by *ASPack*, among others. However, it is probably for an entirely different reason, which is to force a CPU queue flush for multiprocessor environments.

b. Write^Exec

Some unpacking tools work by changing the previously writable-executable page attributes to either writable or executable, but not both. These changes can be detected indirectly. The easier method to achieve this is to use a function that uses the kernel to write to a specified user-mode address. The function will return an error if it fails to write to the address. In this case, the address to specify is one in which the page attributes are likely to have been altered. A good candidate function is the kernel32 VirtualQuery() function.

Example code looks like this:

```
    ;sizeof(MEMORY_BASIC_INFORMATION)
    push 1ch
    mov  ebx, offset l1
    push ebx
    push ebx
    call VirtualQuery
    test eax, eax
    je   being_debugged
    ...
    ;sizeof(MEMORY_BASIC_INFORMATION)
l1: db   1ch dup (?)
```

Not only does the kernel32 VirtualQuery() function write to a specified user-mode address, but it also returns the original value of the page attributes. Any change in the attributes is an indication that an intercepter is running.

Example code looks like this:

```
    ;sizeof(MEMORY_BASIC_INFORMATION)
    push 1ch
    mov  ebx, offset l1
    push ebx
    push ebx
    call VirtualQuery
    ;PAGE_EXECUTE_READWRITE
    cmp  b [ebx+14h], 40h
    jne  being_debugged
    ...
    ;sizeof(MEMORY_BASIC_INFORMATION)
l1: db   1ch dup (?)
```

The kernel32 VirtualProtect() function is another way to query the page attributes, since the previous attributes are returned by the function. Any change in the attributes is an indication that an intercepter is running.

Example code looks like this:

```
l1: push eax
    push esp
    push 40h ;PAGE_EXECUTE_READWRITE
    push 1
    push offset l1
    call VirtualProtect
    pop  eax
    ;PAGE_EXECUTE_READWRITE
    cmp  al, 40h
```

```
        jne  being_debugged
```

V. MISCELLANEOUS

a. Fake signatures

Some packers emit the startup code for other popular packers and tools, in an attempt to fool unpackers into misidentifying the wrapper. Among the most popular of these fake signatures is the startup code for *Microsoft Visual C*, which was written to fool *PEiD*. This technique is used by *RLPack Professional*, among others.

CONCLUSION

There are many different classes of anti-unpacking techniques, and this paper has attempted to describe a subset of the known ones. A companion paper (Anti-Unpacking Tricks - Future) will describe some of the possible future ones, so that we can, where possible, construct defenses against them.

Final note:

The text of this paper was completed before I joined *Microsoft*.
It was produced without access to any *Microsoft* source code or personnel.

[i] http://crackmes.de/users/thehyper/hyperunpackme2/
[ii] http://www.honeynet.org/scans/scan33/index.html
[iii] http://www.securityfocus.com/infocus/1893
[iv] http://www.reactos.org
[v] http://www.piotrbania.com/all/articles/antid.txt
[vi]
http://piotrbania.com/all/articles/bypassing_the_breakpoints.txt
[vii] http://www.defendion.com/EliCZ/infos/TlsInAsm.zip
[viii] http://pferrie.tripod.com/papers/chiton.pdf
[ix] http://vx.eof-project.net/viewtopic.php?id=142
[x] http://pferrie.tripod.com/papers/attacks2.pdf
[xi] http://pferrie.tripod.com/papers/attacks2.pdf
[xii]
http://www.symantec.com/enterprise/security_response/weblog/2007/02/x86_fetchdecode_anomalies.html