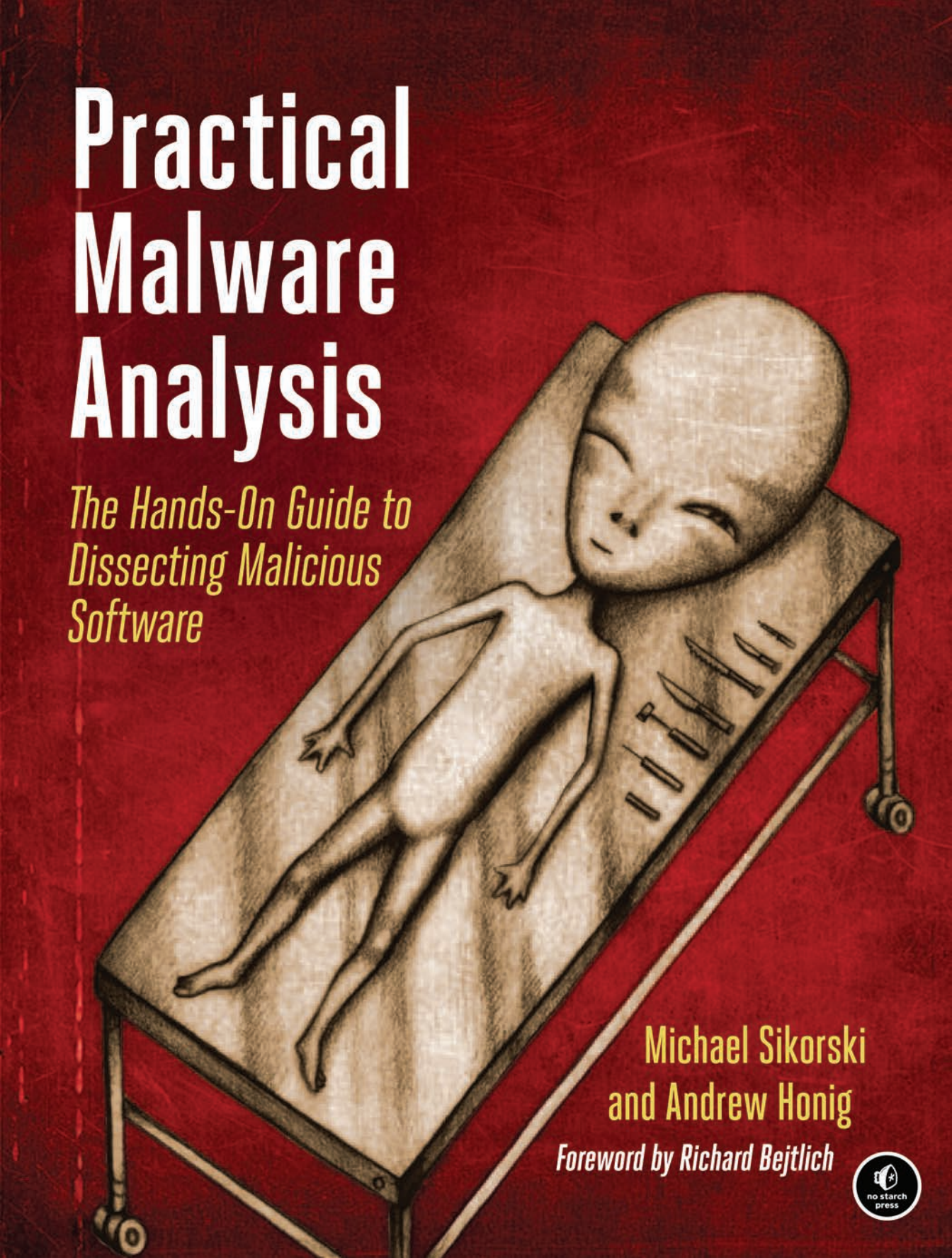
Practical Malware Analysis

The Hands-On Guide to Dissecting Malicious Software



Michael Sikorski
and Andrew Honig

Foreword by Richard Bejtlich

no starch press

BRIEF CONTENTS

12

COVERT MALWARE LAUNCHING

As computer systems and users have become more sophisticated, malware, too, has evolved. For example, because many users know how to list processes with the Windows Task Manager (where malicious software used to appear), malware authors have developed many techniques to blend their malware into the normal Windows landscape, in an effort to conceal it.

This chapter focuses on some of the methods that malware authors use to avoid detection, called *covert launching techniques*. Here, you'll learn how to recognize code constructs and other coding patterns that will help you to identify common ways that malware is covertly launched.

Launchers

As discussed in the previous chapter, a launcher (also known as a *loader*) is a type of malware that sets itself or another piece of malware for immediate or future covert execution. The goal of a launcher is to set up things so that the malicious behavior is concealed from a user.

Launchers often contain the malware that they're designed to load. The most common example is an executable or DLL in its own resource section.

The resource section in the Windows PE file format is used by the executable and is not considered part of the executable. Examples of the normal contents of the resource section include icons, images, menus, and strings. Launchers will often store malware within the resource section. When the launcher is run, it extracts an embedded executable or DLL from the resource section before launching it.

As you have seen in previous examples, if the resource section is compressed or encrypted, the malware must perform resource section extraction before loading. This often means that you will see the launcher use resource-manipulation API functions such as `FindResource`, `LoadResource`, and `SizeofResource`.

Malware launchers often must be run with administrator privileges or escalate themselves to have those privileges. Average user processes can't perform all of the techniques we discuss in this chapter. We discussed privilege escalation in the previous chapter. The fact that launchers may contain privilege-escalation code provides another way to identify them.

Process Injection

The most popular covert launching technique is *process injection*. As the name implies, this technique injects code into another running process, and that process unwittingly executes the malicious code. Malware authors use process injection in an attempt to conceal the malicious behavior of their code, and sometimes they use this to try to bypass host-based firewalls and other process-specific security mechanisms.

Certain Windows API calls are commonly used for process injection. For example, the `VirtualAllocEx` function can be used to allocate space in an external process's memory, and `WriteProcessMemory` can be used to write data to that allocated space. This pair of functions is essential to the first three loading techniques that we'll discuss in this chapter.

DLL Injection

DLL injection—a form of process injection where a remote process is forced to load a malicious DLL—is the most commonly used covert loading technique. DLL injection works by injecting code into a remote process that calls `LoadLibrary`, thereby forcing a DLL to be loaded in the context of that process. Once the compromised process loads the malicious DLL, the OS automatically calls the DLL's `DllMain` function, which is defined by the author of the DLL. This function contains the malicious code and has as much access to the system as the process in which it is running. Malicious DLLs often have little content other than the `Dllmain` function, and everything they do will appear to originate from the compromised process.

Figure 12-1 shows an example of DLL injection. In this example, the launcher malware injects its DLL into Internet Explorer's memory, thereby giving the injected DLL the same access to the Internet as Internet Explorer. The loader malware had been unable to access the Internet prior to injection because a process-specific firewall detected it and blocked it.

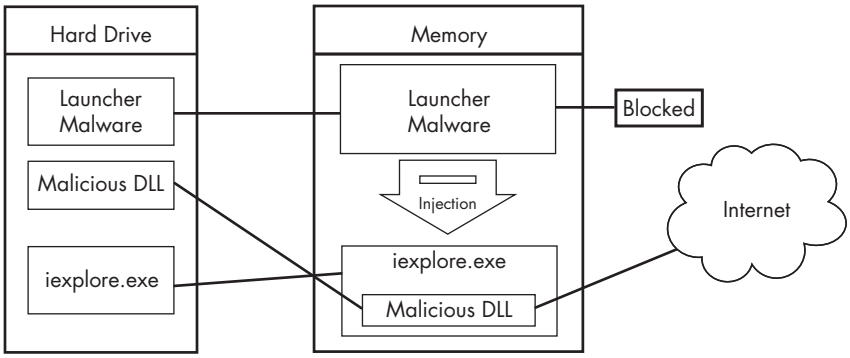


Figure 12-1: DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

In order to inject the malicious DLL into a host program, the launcher malware must first obtain a handle to the victim process. The most common way is to use the Windows API calls `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next` to search the process list for the injection target. Once the target is found, the launcher retrieves the process identifier (PID) of the target process and then uses it to obtain the handle via a call to `OpenProcess`.

The function `CreateRemoteThread` is commonly used for DLL injection to allow the launcher malware to create and execute a new thread in a remote process. When `CreateRemoteThread` is used, it is passed three important parameters: the process handle (`hProcess`) obtained with `OpenProcess`, along with the starting point of the injected thread (`lpStartAddress`) and an argument for that thread (`lpParameter`). For example, the starting point might be set to `LoadLibrary` and the malicious DLL name passed as the argument. This will trigger `LoadLibrary` to be run in the victim process with a parameter of the malicious DLL, thereby causing that DLL to be loaded in the victim process (assuming that `LoadLibrary` is available in the victim process's memory space and that the malicious library name string exists within that same space).

Malware authors generally use `VirtualAllocEx` to create space for the malicious library name string. The `VirtualAllocEx` function allocates space in a remote process if a handle to that process is provided.

The last setup function required before `CreateRemoteThread` can be called is `WriteProcessMemory`. This function writes the malicious library name string into the memory space that was allocated with `VirtualAllocEx`.

Listing 12-1 contains C pseudocode for performing DLL injection.

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID ❶);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName, sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
❷ CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,pNameInVictimProcess,...,...);
```

Listing 12-1: C Pseudocode for DLL injection

This listing assumes that we obtain the victim PID in `victimProcessID` when it is passed to `OpenProcess` at ❶ in order to get the handle to the victim process. Using the handle, `VirtualAllocEx` and `WriteProcessMemory` then allocate space and write the name of the malicious DLL into the victim process. Next, `GetProcAddress` is used to get the address to `LoadLibrary`.

Finally, at ❷, `CreateRemoteThread` is passed the three important parameters discussed earlier: the handle to the victim process, the address of `LoadLibrary`, and a pointer to the malicious DLL name in the victim process. The easiest way to identify DLL injection is by identifying this trademark pattern of Windows API calls when looking at the launcher malware's disassembly.

In DLL injection, the malware launcher never calls a malicious function. As stated earlier, the malicious code is located in `DllMain`, which is automatically called by the OS when the DLL is loaded into memory. The DLL injection launcher's goal is to call `CreateRemoteThread` in order to create the remote thread `LoadLibrary`, with the parameter of the malicious DLL being injected.

Figure 12-2 shows DLL injection code as seen through a debugger. The six function calls from our pseudocode in Listing 12-1 can be seen in the disassembly, labeled ❶ through ❻.

```
004076BB   CALL DWORD PTR DS:[<&KERNEL32.OpenProcess>]      OpenProcess ❶
004076C1   MOV DWORD PTR SS:[EBP-1008],EAX
004076C7   CMP DWORD PTR SS:[EBP-1008],-1
004076CE   JNZ SHORT DLLInjec.004076D8
004076D0   OR EAX,FFFFFFFF
004076D3   JMP DLLInjec.0040779D
004076D8   MOV DWORD PTR SS:[EBP-100C],7D0
004076E2   JMP DLLInjec.00407646
004076E7   PUSH 4
004076E9   PUSH 3000
004076EE   PUSH 104
004076F3   PUSH 0
004076F5   MOV EAX,DWORD PTR SS:[EBP-1008]
004076FB   PUSH EAX
004076FC   CALL DWORD PTR DS:[<&KERNEL32.VirtualAllocEx>]   kernel32.VirtualAllocEx ❷
00407702   MOV DWORD PTR SS:[EBP-1010],EAX
00407708   CMP DWORD PTR SS:[EBP-1010],0
0040770F   JNZ SHORT DLLInjec.00407719
00407711   OR EAX,FFFFFFFF
00407714   JMP DLLInjec.0040779D
00407719   PUSH 0                                          pBytesWritten = NULL
0040771B   PUSH 104                                        BytesToWrite = 104 (260.)
00407720   LEA ECX,DWORD PTR SS:[EBP-1100]
00407726   PUSH ECX                                        Buffer
00407727   MOV EDX,DWORD PTR SS:[EBP-1010]
0040772D   PUSH EDX                                        Address
0040772E   MOV EAX,DWORD PTR SS:[EBP-1008]
00407734   PUSH EAX                                        hProcess
00407735   CALL DWORD PTR DS:[<&KERNEL32.WriteProcessMemory>]  WriteProcessMemory ❸
0040773B   PUSH DLLInjec.0040ACCC                          pModule = "kernel32.dll"
00407740   CALL DWORD PTR DS:[<&KERNEL32.GetModuleHandleW>]  GetModuleHandleW ❹
00407746   MOV DWORD PTR SS:[EBP-1188],EAX
0040774C   PUSH DLLInjec.0040ACE8                          ProcNameOrOrdinal = "LoadLibraryA"
00407751   MOV ECX,DWORD PTR SS:[EBP-1188]
00407757   PUSH ECX                                        hModule
00407758   CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]  GetProcAddress ❺
0040775E   MOV DWORD PTR SS:[EBP-1190],EAX
00407764   PUSH 0
00407766   PUSH 0
00407768   MOV EDX,DWORD PTR SS:[EBP-1010]
0040776E   PUSH EDX
0040776F   MOV EAX,DWORD PTR SS:[EBP-1190]
00407775   PUSH EAX
00407776   PUSH 0
00407778   PUSH 0
0040777A   MOV ECX,DWORD PTR SS:[EBP-1008]
00407780   PUSH ECX
00407781   CALL DWORD PTR DS:[<&KERNEL32.CreateRemoteThread>]  kernel32.CreateRemoteThread ❻
```

Figure 12-2: DLL injection debugger view

Once you find DLL injection activity in disassembly, you should start looking for the strings containing the names of the malicious DLL and the victim process. In the case of Figure 12-2, we don't see those strings, but they must be accessed before this code executes. The victim process name can often be found in a `strncmp` function (or equivalent) when the launcher

determines the victim process's PID. To find the malicious DLL name, we could set a breakpoint at 0x407735 and dump the contents of the stack to reveal the value of `Buffer` as it is being passed to `WriteProcessMemory`.

Once you're able to recognize the DLL injection code pattern and identify these important strings, you should be able to quickly analyze an entire group of malware launchers.

Direct Injection

Like DLL injection, *direct injection* involves allocating and inserting code into the memory space of a remote process. Direct injection uses many of the same Windows API calls as DLL injection. The difference is that instead of writing a separate DLL and forcing the remote process to load it, direct-injection malware injects the malicious code directly into the remote process.

Direct injection is more flexible than DLL injection, but it requires a lot of customized code in order to run successfully without negatively impacting the host process. This technique can be used to inject compiled code, but more often, it's used to inject shellcode.

Three functions are commonly found in cases of direct injection: `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. There will typically be two calls to `VirtualAllocEx` and `WriteProcessMemory`. The first will allocate and write the data used by the remote thread, and the second will allocate and write the remote thread code. The call to `CreateRemoteThread` will contain the location of the remote thread code (`lpStartAddress`) and the data (`lpParameter`).

Since the data and functions used by the remote thread must exist in the victim process, normal compilation procedures will not work. For example, strings are not in the normal `.data` section, and `LoadLibrary/GetProcAddress` will need to be called to access functions that are not already loaded. There are other restrictions, which we won't go into here. Basically, direct injection requires that authors either be skilled assembly language coders or that they will inject only relatively simple shellcode.

In order to analyze the remote thread's code, you may need to debug the malware and dump all memory buffers that occur before calls to `WriteProcessMemory` to be analyzed in a disassembler. Since these buffers most often contain shellcode, you will need shellcode analysis skills, which we discuss extensively in Chapter 19.

Process Replacement

Rather than inject code into a host program, some malware uses a method known as *process replacement* to overwrite the memory space of a running process with a malicious executable. Process replacement is used when a malware author wants to disguise malware as a legitimate process, without the risk of crashing a process through the use of process injection.

This technique provides the malware with the same privileges as the process it is replacing. For example, if a piece of malware were to perform a process-replacement attack on *svchost.exe*, the user would see a process

name *svchost.exe* running from *C:\Windows\System32* and probably think nothing of it. (This is a common malware attack, by the way.)

Key to process replacement is creating a process in a *suspended state*. This means that the process will be loaded into memory, but the primary thread of the process is suspended. The program will not do anything until an external program resumes the primary thread, causing the program to start running. Listing 12-2 shows how a malware author achieves this suspended state by passing `CREATE_SUSPENDED` (`0x4`) as the `dwCreationFlags` parameter when performing the call to `CreateProcess`.

```
00401535        push    edi             ; lpProcessInformation
00401536        push    ecx             ; lpStartupInfo
00401537        push    ebx             ; lpCurrentDirectory
00401538        push    ebx             ; lpEnvironment
00401539        push    CREATE_SUSPENDED ; dwCreationFlags
0040153B        push    ebx             ; bInheritHandles
0040153C        push    ebx             ; lpThreadAttributes
0040153D        lea     edx, [esp+94h+CommandLine]
00401541        push    ebx             ; lpProcessAttributes
00401542        push    edx             ; lpCommandLine
00401543        push    ebx             ; lpApplicationName
00401544        mov     [esp+0A0h+StartupInfo.dwFlags], 101h
0040154F        mov     [esp+0A0h+StartupInfo.wShowWindow], bx
00401557        call    ds:CreateProcessA
```

Listing 12-2: Assembly code showing process replacement

Although poorly documented by Microsoft, this method of process creation can be used to load a process into memory and suspend it at the entry point.

Listing 12-3 shows C pseudocode for performing process replacement.

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
 ❶ WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

Listing 12-3: C pseudocode for process replacement

Once the process is created, the next step is to replace the victim process's memory with the malicious executable, typically using `ZwUnmapViewOfSection` to release all memory pointed to by a section passed as a parameter. After the memory is unmapped, the loader performs `VirtualAllocEx` to allocate

new memory for the malware, and uses `WriteProcessMemory` to write each of the malware sections to the victim process space, typically in a loop, as shown at ❶.

In the final step, the malware restores the victim process environment so that the malicious code can run by calling `SetThreadContext` to set the entry point to point to the malicious code. Finally, `ResumeThread` is called to initiate the malware, which has now replaced the victim process.

Process replacement is an effective way for malware to appear non-malicious. By masquerading as the victim process, the malware is able to bypass firewalls or intrusion prevention systems (IPSs) and avoid detection by appearing to be a normal Windows process. Also, by using the original binary's path, the malware deceives the savvy user who, when viewing a process listing, sees only the known and valid binary executing, with no idea that it was unmapped.

Hook Injection

Hook injection describes a way to load malware that takes advantage of Windows *hooks*, which are used to intercept messages destined for applications. Malware authors can use hook injection to accomplish two things:

- To be sure that malicious code will run whenever a particular message is intercepted
- To be sure that a particular DLL will be loaded in a victim process's memory space

As shown in Figure 12-3, users generate events that are sent to the OS, which then sends messages created by those events to threads registered to receive them. The right side of the figure shows one way that an attacker can insert a malicious DLL to intercept messages.

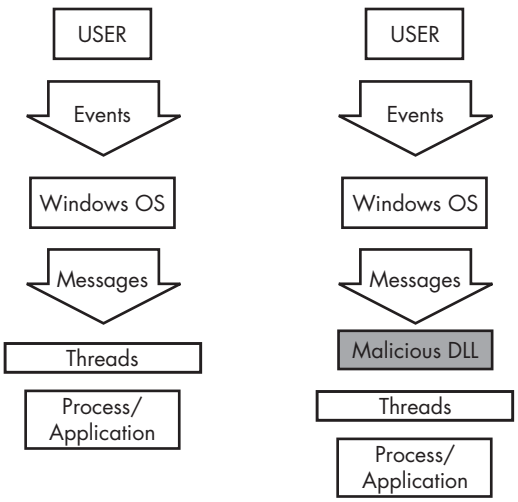


Figure 12-3: Event and message flow in Windows with and without hook injection

Local and Remote Hooks

There are two types of Windows hooks:

- *Local hooks* are used to observe or manipulate messages destined for an internal process.
- *Remote hooks* are used to observe or manipulate messages destined for a remote process (another process on the system).

Remote hooks are available in two forms: high and low level. High-level remote hooks require that the hook procedure be an exported function contained in a DLL, which will be mapped by the OS into the process space of a hooked thread or all threads. Low-level remote hooks require that the hook procedure be contained in the process that installed the hook. This procedure is notified before the OS gets a chance to process the event.

Keyloggers Using Hooks

Hook injection is frequently used in malicious applications known as *keyloggers*, which record keystrokes. Keystrokes can be captured by registering high- or low-level hooks using the `WH_KEYBOARD` or `WH_KEYBOARD_LL` hook procedure types, respectively.

For `WH_KEYBOARD` procedures, the hook will often be running in the context of a remote process, but it can also run in the process that installed the hook. For `WH_KEYBOARD_LL` procedures, the events are sent directly to the process that installed the hook, so the hook will be running in the context of the process that created it. Using either hook type, a keylogger can intercept keystrokes and log them to a file or alter them before passing them along to the process or system.

Using SetWindowsHookEx

The principal function call used to perform remote Windows hooking is `SetWindowsHookEx`, which has the following parameters:

`idHook` Specifies the type of hook procedure to install.

`lpfn` Points to the hook procedure.

`hMod` For high-level hooks, identifies the handle to the DLL containing the hook procedure defined by `lpfn`. For low-level hooks, this identifies the local module in which the `lpfn` procedure is defined.

`dwThreadId` Specifies the identifier of the thread with which the hook procedure is to be associated. If this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread. This must be set to zero for low-level hooks.

The hook procedure can contain code to process messages as they come in from the system, or it can do nothing. Either way, the hook procedure must call `CallNextHookEx`, which ensures that the next hook procedure in the call chain gets the message and that the system continues to run properly.

Thread Targeting

When targeting a specific `dwThreadId`, malware generally includes instructions for determining which system thread identifier to use, or it is designed to load into all threads. That said, malware will load into all threads only if it's a keylogger or the equivalent (when the goal is message interception). However, loading into all threads can degrade the running system and may trigger an IPS. Therefore, if the goal is to simply load a DLL in a remote process, only a single thread will be injected in order to remain stealthy.

Targeting a single thread requires a search of the process listing for the target process and can require that the malware run a program if the target process is not already running. If a malicious application hooks a Windows message that is used frequently, it's more likely to trigger an IPS, so malware will often set a hook with a message that is not often used, such as `WH_CBT` (a computer-based training message).

Listing 12-4 shows the assembly code for performing hook injection in order to load a DLL in a different process's memory space.

```
00401100        push    esi
00401101        push    edi
00401102        push    offset LibFileName ; "hook.dll"
00401107        call    LoadLibraryA
0040110D        mov     esi, eax
0040110F        push    offset ProcName ; "MalwareProc"
00401114        push    esi             ; hModule
00401115        call    GetProcAddress
0040111B        mov     edi, eax
0040111D        call    GetNotepadThreadId
00401122        push    eax             ; dwThreadId
00401123        push    esi             ; hmod
00401124        push    edi             ; lpfn
00401125        push    WH_CBT   ; idHook
00401127        call    SetWindowsHookExA
```

Listing 12-4: Hook injection, assembly code

In Listing 12-4, the malicious DLL (*hook.dll*) is loaded by the malware, and the malicious hook procedure address is obtained. The hook procedure, `MalwareProc`, calls only `CallNextHookEx`. `SetWindowsHookEx` is then called for a thread in *notepad.exe* (assuming that *notepad.exe* is running). `GetNotepadThreadId` is a locally defined function that obtains a `dwThreadId` for *notepad.exe*. Finally, a `WH_CBT` message is sent to the injected *notepad.exe* in order to force *hook.dll* to be loaded by *notepad.exe*. This allows *hook.dll* to run in the *notepad.exe* process space.

Once *hook.dll* is injected, it can execute the full malicious code stored in `DllMain`, while disguised as the *notepad.exe* process. Since `MalwareProc` calls only `CallNextHookEx`, it should not interfere with incoming messages, but malware often immediately calls `LoadLibrary` and `UnhookWindowsHookEx` in `DllMain` to ensure that incoming messages are not impacted.

Detours

Detours is a library developed by Microsoft Research in 1999. It was originally intended as a way to easily instrument and extend existing OS and application functionality. The Detours library makes it possible for a developer to make application modifications simply.

Malware authors like Detours, too, and they use the Detours library to perform import table modification, attach DLLs to existing program files, and add function hooks to running processes.

Malware authors most commonly use Detours to add new DLLs to existing binaries on disk. The malware modifies the PE structure and creates a section named `.detour`, which is typically placed between the export table and any debug symbols. The `.detour` section contains the original PE header with a new import address table. The malware author then uses Detours to modify the PE header to point to the new import table, by using the `setdll` tool provided with the Detours library.

Figure 12-4 shows a PEview of Detours being used to trojanize *notepad.exe*. Notice in the `.detour` section at ❶ that the new import table contains *evil.dll*, seen at ❷. *Evil.dll* will now be loaded whenever Notepad is launched. Notepad will continue to operate as usual, and most users would have no idea that the malicious DLL was executed.

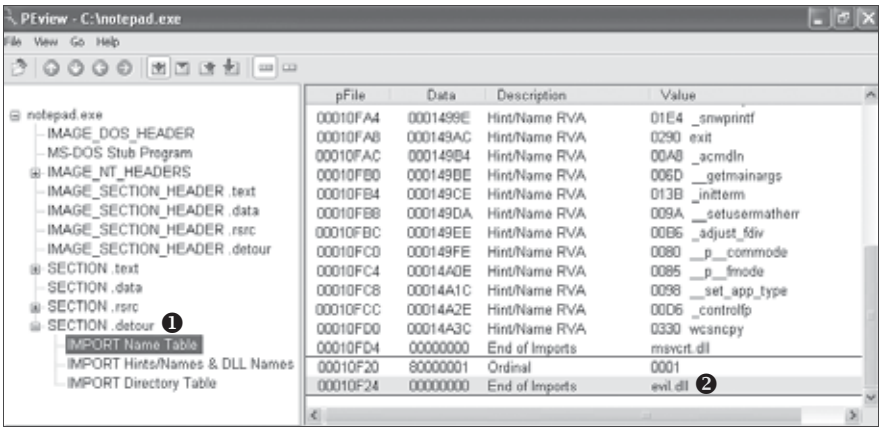


Figure 12-4: A PEview of Detours and the evil.dll

Instead of using the official Microsoft Detours library, malware authors have been known to use alternative and custom methods to add a `.detour` section. The use of these methods for detour addition should not impact your ability to analyze the malware.

APC Injection

Earlier in this chapter, you saw that by creating a thread using `CreateRemoteThread`, you can invoke functionality in a remote process. However, thread creation requires overhead, so it would be more efficient to invoke a function on

an existing thread. This capability exists in Windows as the *asynchronous procedure call (APC)*.

APCs can direct a thread to execute some other code prior to executing its regular execution path. Every thread has a queue of APCs attached to it, and these are processed when the thread is in an alertable state, such as when they call functions like `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx`, and `Sleep`. These functions essentially give the thread a chance to process the waiting APCs.

If an application queues an APC while the thread is alertable but before the thread begins running, the thread begins by calling the APC function. A thread calls the APC functions one by one for all APCs in its APC queue. When the APC queue is complete, the thread continues running along its regular execution path. Malware authors use APCs to preempt threads in an alertable state in order to get immediate execution for their code.

APCs come in two forms:

- An APC generated for the system or a driver is called a *kernel-mode APC*.
- An APC generated for an application is called a *user-mode APC*.

Malware generates user-mode APCs from both kernel and user space using *APC injection*. Let's take a closer look at each of these methods.

APC Injection from User Space

From user space, another thread can queue a function to be invoked in a remote thread, using the API function `QueueUserAPC`. Because a thread must be in an alertable state in order to run a user-mode APC, malware will look to target threads in processes that are likely to go into that state. Luckily for the malware analyst, `WaitForSingleObjectEx` is the most common call in the Windows API, and there are usually many threads in the alertable state.

Let's examine the `QueueUserAPC`'s parameters: `pfnAPC`, `hThread`, and `dwData`. A call to `QueueUserAPC` is a request for the thread whose handle is `hThread` to run the function defined by `pfnAPC` with the parameter `dwData`. Listing 12-5 shows how malware can use `QueueUserAPC` to force a DLL to be loaded in the context of another process, although before we arrive at this code, the malware has already picked a target thread.

NOTE *During analysis, you can find thread-targeting code by looking for API calls such as `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next` for the malware to find the target process. These API calls will often be followed by calls to `Thread32First` and `Thread32Next`, which will be in a loop looking to target a thread contained in the target process. Alternatively, malware can also use `Nt/ZwQuerySystemInformation` with the `SYSTEM_PROCESS_INFORMATION` information class to find the target process.*

```
00401DA9        push    [esp+4+dwThreadId]      ; dwThreadId
00401DAD        push    0                       ; bInheritHandle
00401DAF        push    10h                     ; dwDesiredAccess
00401DB1        call    ds:OpenThread ❶
00401DB7        mov     esi, eax
```

```
00401DB9        test    esi, esi
00401DBB        jz      short loc_401DCE
00401DBD        push    [esp+4+dwData]          ; dwData = dbnet.dll
00401DC1        push    esi                     ; hThread
00401DC2        push    ds:LoadLibraryA ❷       ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

Listing 12-5: APC injection from a user-mode application

Once a target-thread identifier is obtained, the malware uses it to open a handle to the thread, as seen at ❶. In this example, the malware is looking to force the thread to load a DLL in the remote process, so you see a call to `QueueUserAPC` with the `pfnAPC` set to `LoadLibraryA` at ❷. The parameter to be sent to `LoadLibraryA` will be contained in `dwData` (in this example, that was set to the DLL *dbnet.dll* earlier in the code). Once this APC is queued and the thread goes into an alertable state, `LoadLibraryA` will be called by the remote thread, causing the target process to load *dbnet.dll*.

In this example, the malware targeted *svchost.exe*, which is a popular target for APC injection because its threads are often in an alertable state. Malware may APC-inject into every thread of *svchost.exe* just to ensure that execution occurs quickly.

APC Injection from Kernel Space

Malware drivers and rootkits often wish to execute code in user space, but there is no easy way for them to do it. One method they use is to perform APC injection from kernel space to get their code execution in user space. A malicious driver can build an APC and dispatch a thread to execute it in a user-mode process (most often *svchost.exe*). APCs of this type often consist of shellcode.

Device drivers leverage two major functions in order to utilize APCs: `KeInitializeApc` and `KeInsertQueueApc`. Listing 12-6 shows an example of these functions in use in a rootkit.

```
000119BD        push    ebx
000119BE        push    1 ❶
000119C0        push    [ebp+arg_4] ❷
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0] ❸
000119CE        push    esi
000119CF        call    ds:KeInitializeApc
000119D5        cmp     edi, ebx
000119D7        jz      short loc_119EA
000119D9        push    ebx
000119DA        push    [ebp+arg_C]
000119DD        push    [ebp+arg_8]
000119E0        push    esi
000119E1        call    edi       ;KeInsertQueueApc
```

Listing 12-6: User-mode APC injection from kernel space

The APC first must be initialized with a call to `KeInitializeApc`. If the sixth parameter (`NormalRoutine`) ❷ is non-zero in combination with the seventh parameter (`ApcMode`) ❶ being set to 1, then we are looking at a user-mode type. Therefore, focusing on these two parameters can tell you if the rootkit is using APC injection to run code in user space.

`KeInitializeAPC` initializes a KAPC structure, which must be passed to `KeInsertQueueApc` to place the APC object in the target thread's corresponding APC queue. In Listing 12-6, ESI will contain the KAPC structure. Once `KeInsertQueueApc` is successful, the APC will be queued to run.

In this example, the malware targeted *svchost.exe*, but to make that determination, we would need to trace back the second-to-last parameter pushed on the stack to `KeInitializeApc`. This parameter contains the thread that will be injected. In this case, it is contained in `arg_0`, as seen at ❸. Therefore, we would need to look back in the code to check how `arg_0` was set in order to see that *svchost.exe*'s threads were targeted.

Conclusion

In this chapter, we've explored the common covert methods through which malware launches, ranging from the simple to advanced. Many of the techniques involve manipulating live memory on the system, as with DLL injection, process replacement, and hook injection. Other techniques involve modifying binaries on disk, as in the case of adding a `.detour` section to a PE file. Although these techniques are all very different, they achieve the same goal.

A malware analyst must be able to recognize launching techniques in order to know how to find malware on a live system. Recognizing and analyzing launching techniques is really only part of the full analysis, since all launchers do only one thing: they get the malware running.

In the next two chapters, you will learn how malware encodes its data and communicates over the network.

LABS

Lab 12-1

Analyze the malware found in the file *Lab12-01.exe* and *Lab12-01.dll*. Make sure that these files are in the same directory when performing the analysis.

Questions

1. What happens when you run the malware executable?
2. What process is being injected?
3. How can you make the malware stop the pop-ups?
4. How does this malware operate?

Lab 12-2

Analyze the malware found in the file *Lab12-02.exe*.

Questions

1. What is the purpose of this program?
2. How does the launcher program hide execution?
3. Where is the malicious payload stored?
4. How is the malicious payload protected?
5. How are strings protected?

Lab 12-3

Analyze the malware extracted during the analysis of Lab 12-2, or use the file *Lab12-03.exe*.

Questions

1. What is the purpose of this malicious payload?
2. How does the malicious payload inject itself?
3. What filesystem residue does this program create?

Lab 12-4

Analyze the malware found in the file *Lab12-04.exe*.

Questions

1. What does the code at 0x401000 accomplish?
2. Which process has code injected?
3. What DLL is loaded using `LoadLibraryA`?
4. What is the fourth argument passed to the `CreateRemoteThread` call?
5. What malware is dropped by the main executable?
6. What is the purpose of this and the dropped malware?