Covert Malware Launching

CYBR473 – Malware and Reverse Engineering (2024/T1)

Lecturers: Arman Khouzani, Alvin Valera

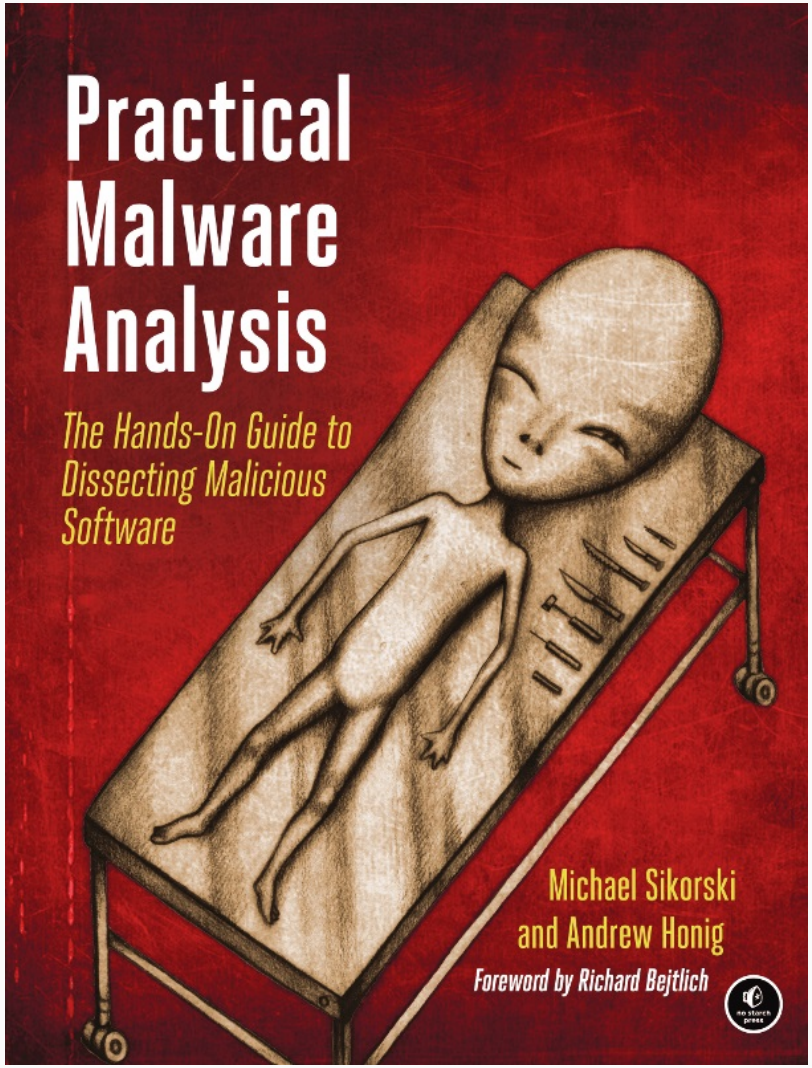
Victoria University of Wellington – School of Engineering and Computer Science

Table of contents

Reading Material

- Part IV: Malware Functionality
  - Ch.12: Covert Malware Launching

“Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software”, Michael Sikorski and Andrew Honig, 2012



Objectives

A malware analyst must be able to recognise launching techniques in order to know how to find malware on a live system

- Describe common techniques for adding malicious functionality to programs
- Recognise how this allows malware to hide itself

Launcher

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

Launcher

Launcher (a.k.a **Loader**) is a type of malware that sets itself or another piece of malware for immediate or future covert execution

Launchers often contain the malware that they're designed to load.

- The most common example is an executable or DLL (the real payload) embedded in its own resource section (`.rsrc`).
  - Examples of the normal contents of the resource section include icons, images, menus, and strings
- If the resource section is compressed or encrypted, the malware must perform resource section extraction before loading.
- This often means that the launcher uses resource-manipulation API functions such as `FindResource`, `LoadResource`, and `SizeofResource`.

Launcher

```
HRSRC FindResourceA(
  [in, optional] HMODULE hModule,
  [in]           LPCSTR  lpName,
  [in]           LPCSTR  lpType
);
```

Determines the location of a resource with the specified type & name in the specified "*module*" (Windows term for an executable or DLL).

Parameters:

- `[in, optional] hModule`
  A handle to the module whose executable file contains the resource. If **`NULL`**, the module used to create the current process.
- `[in] lpName`
  The name of the resource (as a `LPCSTR`, or an integer identifier).
- `[in] lpType`
  The resource type (as a `LPCSTR`, or an integer identifier).

Launcher

```
HGLOBAL LoadResource(
  [in, optional] HMODULE hModule,
  [in]           HRSRC   hResInfo
);
```

Retrieves a handle that can be used to obtain a pointer to the first byte of the specified resource in memory.

Parameters:

- `[in, optional] hModule`
  handle to the module whose executable file contains the resource. Default is the module that created the current process.
- `[in] hResInfo`
  handle to the resource to be loaded, returned by **`FindResource`**.

Launcher

```
DWORD SizeofResource(
  [in, optional] HMODULE hModule,
  [in]           HRSRC   hResInfo
);
```

Retrieves the size, in bytes, of the specified resource.

Parameters:

- `[in, optional] hModule`
  handle to the module whose executable file contains the resource. Default is the module that created the current process.
- `[in] hResInfo`
  handle to the resource to be loaded, returned by **`FindResource`**.

Launcher

Note: 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 that we will introduce in this lecture.

- We discussed privilege escalation previously (the "Malware behaviour" topic).

The fact that launchers may contain privilege-escalation code provides another way to identify them.

Process Injection

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

Process Injection

Process Injection: inject code into another running process, and that process unwittingly executes the malicious code.

- to conceal the malicious behaviour of their code
- to bypass host-based firewalls or process-specific security

Commonly used Windows API calls used in process injection:

- ▷ `VirtualAllocEx`: allocate space in an external process's memory
- ▷ `WriteProcessMemory`: write data to that allocated space.

Essential to these loading techniques that we will discuss:

- ▶ DLL Injection
- ▶ Process Replacement
- ▶ Hook Replacement

Process Injection

```
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

Reserves, commits, or changes (determined by `flAllocationType`) the state of a region of memory within the virtual address space of a process (specified by `hProcess`, which must have the **`PROCESS_VM_OPERATION`** access right.).

If the function succeeds, the return value is the base address of the allocated region of pages. If it fails, returns **`NULL`**.

Process Injection

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

Writes data to an area of memory in a process (specified by `hProcess`, which must have **`PROCESS_VM_WRITE`** and **`PROCESS_VM_OPERATION`** access rights to the process).

The return value is nonzero if it succeeds and 0 if it fails. It fails if the requested write crosses into an inaccessible area of the process.

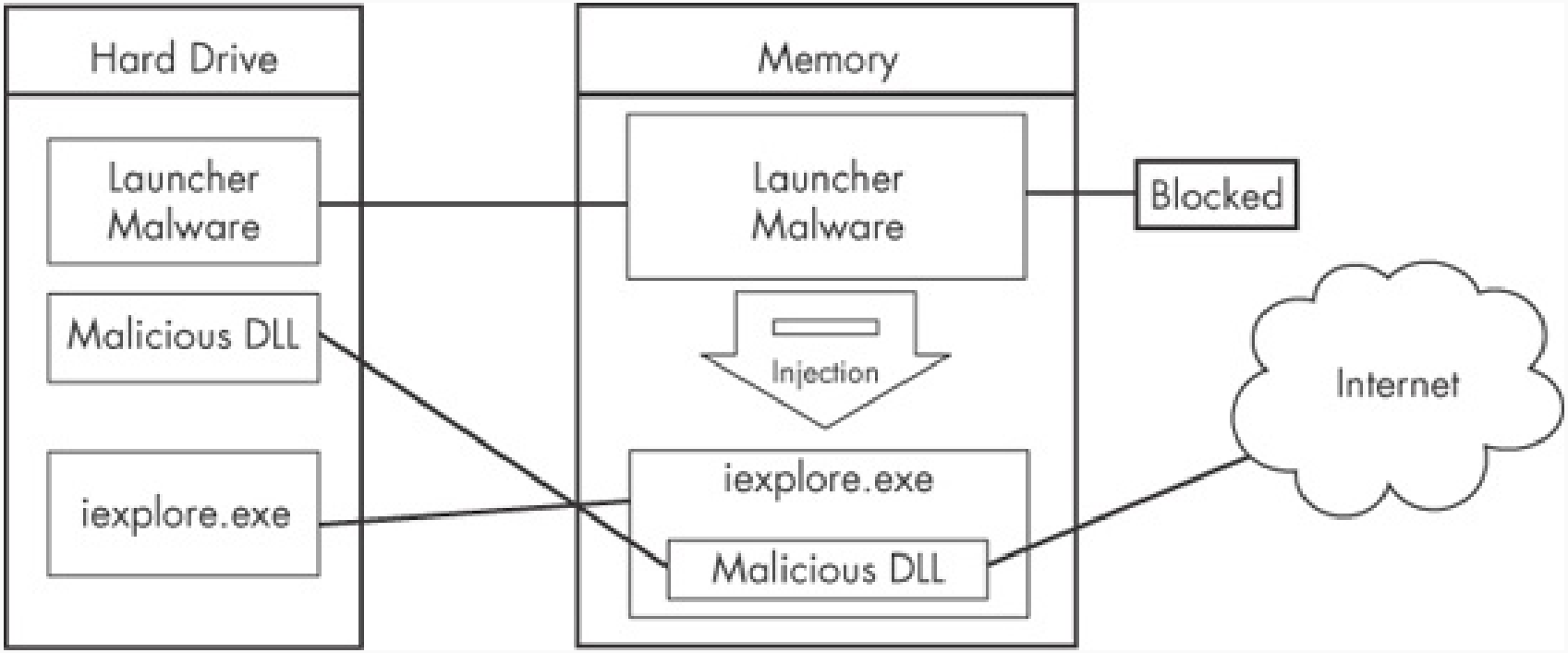
DLL Injection

DLL Injection:

- a remote process is injected with a code containing calls to `LoadLibrary` with a path to a malicious DLL on the disk.
- the OS then loads the DLL and automatically calls its `DllMain` function, which has as much access to the system as the process in which it is running (it is in the same security context).
  - also everything the malicious DLL does will appear to originate from the compromised process.

While DLL injection can have legitimate purposes, e.g. debugging, monitoring or performance analysis, it is a common technique used by malware to evade detection, escalate privileges, or steal data.

DLL Injection



The launcher gets to access the Internet via injecting into `iexplore.exe`.

DLL Injection

```
HMODULE LoadLibraryA(
  [in] LPCSTR lpLibFileName
);
```

Loads the specified module into the address space of the calling process. The module may cause other modules to be loaded.

Parameters:

- `[in] lpLibFileName`
  The (full-path of the) file name of the (DLL) module

If the function succeeds, the return value is a handle to the module.

DLL Injection

For DLL injection, the launcher malware must first obtain a **handle** to the victim process:

- Use the Windows API calls **`CreateToolhelp32Snapshot`**, **`Process32First`**, and **`Process32Next`** to search the process list for the injection target.
- Once the target process **PID** is found, use it to obtain the handle via a call to **`OpenProcess`**

Then **`CreateRemoteThread`** is commonly used to create and execute a thread in the target process, passed 3 main parameters:

- the process handle (`hProcess`) obtained with **`OpenProcess`**
- the starting point of the injected thread (`lpStartAddress`), e.g. set to **`LoadLibrary`**
- an argument for that thread (`lpParameter`), e.g. name of the malicious DLL.

DLL Injection

```
HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

Takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes.

If the function succeeds, it returns an open handle to the specified snapshot. If it fails, returns `INVALID_HANDLE_VALUE`.

DLL Injection

```
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

Parameters:

- `[in] hSnapshot`
  A handle to the snapshot returned from a previous call to the **`CreateToolhelp32Snapshot`** function.
- `[in, out] lppe`
  A pointer to a **`PROCESSENTRY32`** structure. It contains process information such as the name of the executable file, the process identifier, and the process identifier of the parent process.

DLL Injection

```
typedef struct tagPROCESSENTRY32 {
  DWORD      dwSize;
  DWORD      cntUsage;
  DWORD      th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD      th32ModuleID;
  DWORD      cntThreads;
  DWORD      th32ParentProcessID;
  LONG       pcPriClassBase;
  DWORD      dwFlags;
  CHAR       szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

DLL Injection

```
BOOL Process32Next(
  [in]  HANDLE            hSnapshot,
  [out] LPPROCESSENTRY32 lppe
);
```

Retrieves information about the next process recorded in a system snapshot.

Return Value:

- `TRUE` if the next entry of the process list has been copied to the buffer or `FALSE` otherwise. The `ERROR_NO_MORE_FILES` error value is returned by the `GetLastError` function if no processes exist or the snapshot does not contain process information.

DLL Injection

```
HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

Opens an existing local process object.

Important Parameters:

- `[in] dwDesiredAccess`
  The access to the process object. This access right is checked against the security descriptor for the process.
- `[in] dwProcessId`
  The identifier of the local process to be opened. If **`SeDebugPrivilege`** is enabled, all rights will be granted.

If succeeds, returns an open handle to the process, otherwise, **`NULL`**.

DLL Injection

```
HANDLE CreateRemoteThread(
  [in]  HANDLE                  hProcess,
  [in]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]  SIZE_T                  dwStackSize,
  [in]  LPTHREAD_START_ROUTINE  lpStartAddress,
  [in]  LPVOID                  lpParameter,
  [in]  DWORD                   dwCreationFlags,
  [out] LPDWORD                 lpThreadId
);
```

Creates a thread that runs in the virtual address space of another process. The thread has access to all objects that the process opens.

If succeeds, returns a handle to the new thread, **`NULL`** if fails.

DLL Injection

Main Parameters:

`[in] hProcess`

A handle to the process in which the thread is to be created. The handle must have the **`PROCESS_CREATE_THREAD`**, **`PROCESS_QUERY_INFORMATION`**, **`PROCESS_VM_OPERATION`**, **`PROCESS_VM_WRITE`**, and **`PROCESS_VM_READ`** access rights.

`[in] lpStartAddress`

A pointer to the application-defined function to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process.

`[in] lpParameter`

A pointer to a variable to be passed to the thread function.

DLL Injection

```
FARPROC GetProcAddress(
  [in] HMODULE hModule,
  [in] LPCSTR  lpProcName
);
```

Retrieves the address of an exported function (also known as a procedure) or variable from the specified dynamic-link library (DLL).

Parameters:

- `[in] hModule`
  A handle to the DLL module that contains the function or variable, e.g. returned by **`GetModuleHandle`** function.
- `[in] lpProcName`
  The function or variable name, or the function's ordinal value.

Returns address of the exported function or variable, or **`NULL`** if fails.

DLL Injection

Recall that for DLL injection, **`CreateRemoteThread`** is called with 3 main parameters:

- the process handle (`hProcess`) obtained with **`OpenProcess`**
- the starting point of the injected thread (`lpStartAddress`) set to the **`LoadLibrary`** function, obtained via **`GetProcAddress`**.
- an argument for that thread (`lpParameter`), i.e., the name of the malicious DLL.

This assumes that **`LoadLibrary`** function and the malicious library name string exist within the victim process's memory space. **`LoadLibrary`** is in **`kernel32`**, which is always available. For the latter, malware uses **`VirtualAllocEx`** and **`WriteProcessMemory`**:

- **`VirtualAllocEx`** to allocate space in the target process if a handle to that process is provided.
- **`WriteProcessMemory`** to write the malicious library name string into that allocated memory space.

DLL Injection

C Pseudocode for performing DLL injection:

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID);
remoteBuffer = VirtualAllocEx(hVictimProcess, NULL, sizeof malicdllPath,
    MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(hVictimProcess, remoteBuffer, (LPVOID)malicdllPath,
    sizeof malicdllPath, NULL);
threatStartRoutineAddress =
    GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");

CreateRemoteThread(hVictimProcess, NULL, 0, threatStartRoutineAddress,
    remoteBuffer, 0, NULL);
CloseHandle(hVictimProcess);
```

The easiest way to identify DLL injection is by identifying this trademark pattern of Windows API calls in disassembly.

In DLL injection, the launcher never calls a malicious function: the malicious code is located in **`DllMain`**, which is automatically called by the OS when the DLL is loaded by the **`LoadLibrary`** thread.

DLL Injection

DLL injection viewed in a debugger. Function calls are labelled ❶–❻:

```
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-1180]
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
```

DLL Injection

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.

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`.

Direct Injection

Direct Injection:

Like DLL injection, but 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, but requires a lot of customized code in to run successfully without negatively impacting the host process.

This can be used to inject compiled code, but more often, shellcode.

There will typically be two calls to `VirtualAllocEx` and `WriteProcessMemory`:

- The first allocates and writes the **data** used by the remote thread.
- The second allocates and writes the remote thread **code**.

The call to `CreateRemoteThread` contains location of the remote thread code (`lpStartAddress`) and the data (`lpParameter`).

Direct Injection

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.

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.

Process Hollowing

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

Process Replacement (Process Hollowing)

Rather than inject code into a host program, a method known as **process replacement** overwrites the memory space of a running process with a malicious executable, providing it with the same privileges as the process it is replacing.

This is used when a malware wants to disguise as a legitimate process, without the risk of crashing it through process injection.

Key to process replacement is creating a process in a **suspended state**: 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.

Process Hollowing

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

One of the possible values for `dwCreationFlags`:

`CREATE_SUSPENDED` (**`0x4`**): The primary thread of the new process is created in a suspended state, and does not run until the **`ResumeThread`** function is called.

Process Hollowing

The following example shows how this suspended state is achieved by passing **`CREATE_SUSPENDED`** (**`0x4`**) for 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
```

Process Hollowing

The next example shows a C pseudocode for process replacement.

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

Process Hollowing

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 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 in the example.

Next, 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 Hollowing

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

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

Hook Injection

Hook injection:

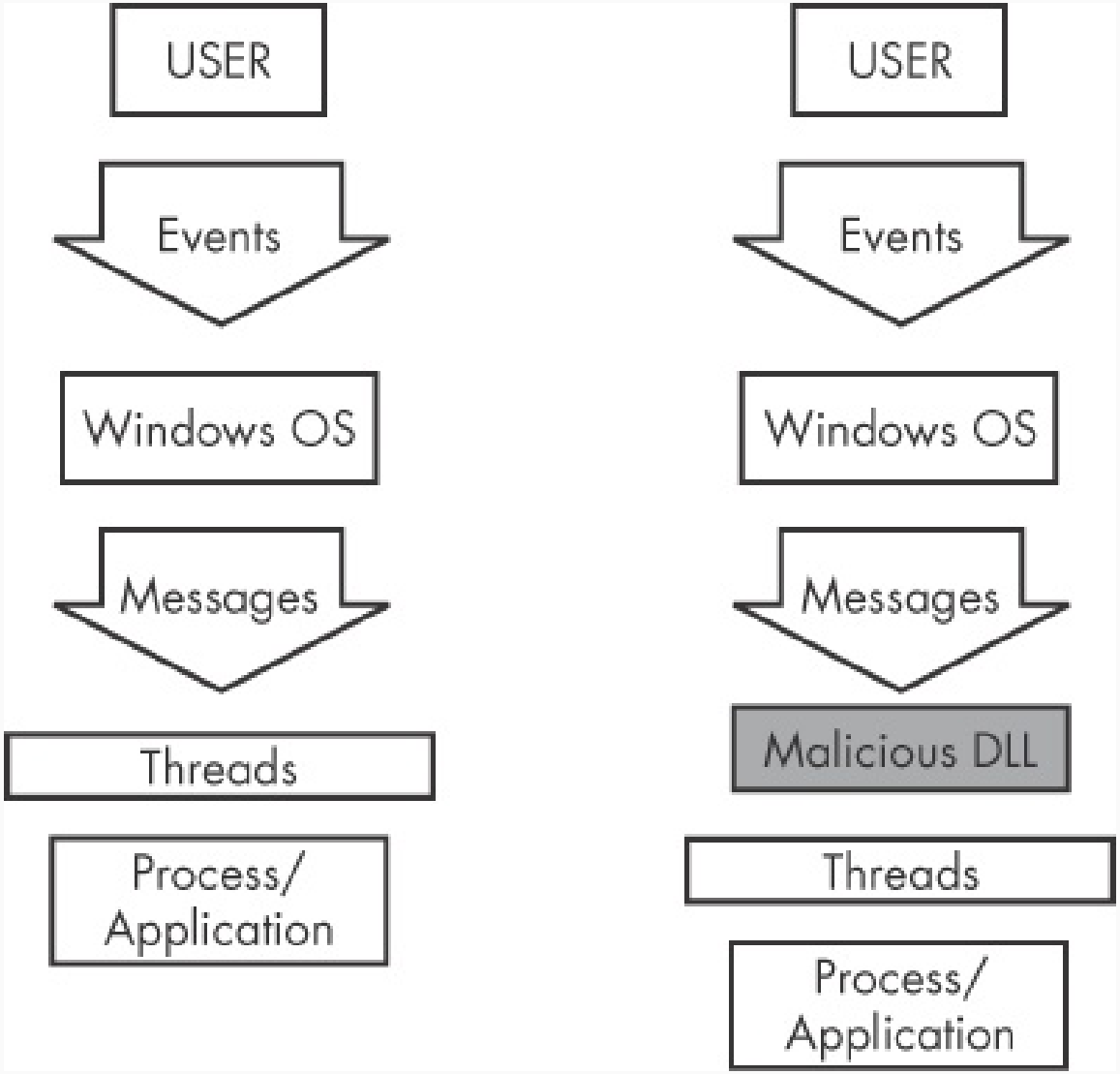
A method to load malware that takes advantage of Windows hooks, which are used to intercept messages destined for applications.

Malware authors use hook injection to accomplish two things:

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

Users generate events that are sent to the OS, which then sends messages created by those events to threads registered to receive them. An attacker can insert a malicious DLL to intercept them.

Hook Injection



Event and message flow in Windows with and without hook injection

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-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.

Hook Injection: 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 through.

Hook Injection: 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, the handle to the DLL containing the hook procedure defined by **`lpfn`**. For low-level hooks, the local module in which the **`lpfn`** procedure is defined.
- **`dwThreadId`.** the identifier of the thread with which the hook procedure is to be associated. If zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread. This must be zero for low-level hooks.

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.

Hook Injection: 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.

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, to remain stealthy.

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).

Hook Injection: Thread Targeting

The following example 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
```

Hook Injection: Thread Targeting

In the previous example, 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 it 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`** to force **`hook.dll`** to be loaded by **`notepad.exe`**.

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

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

Detours

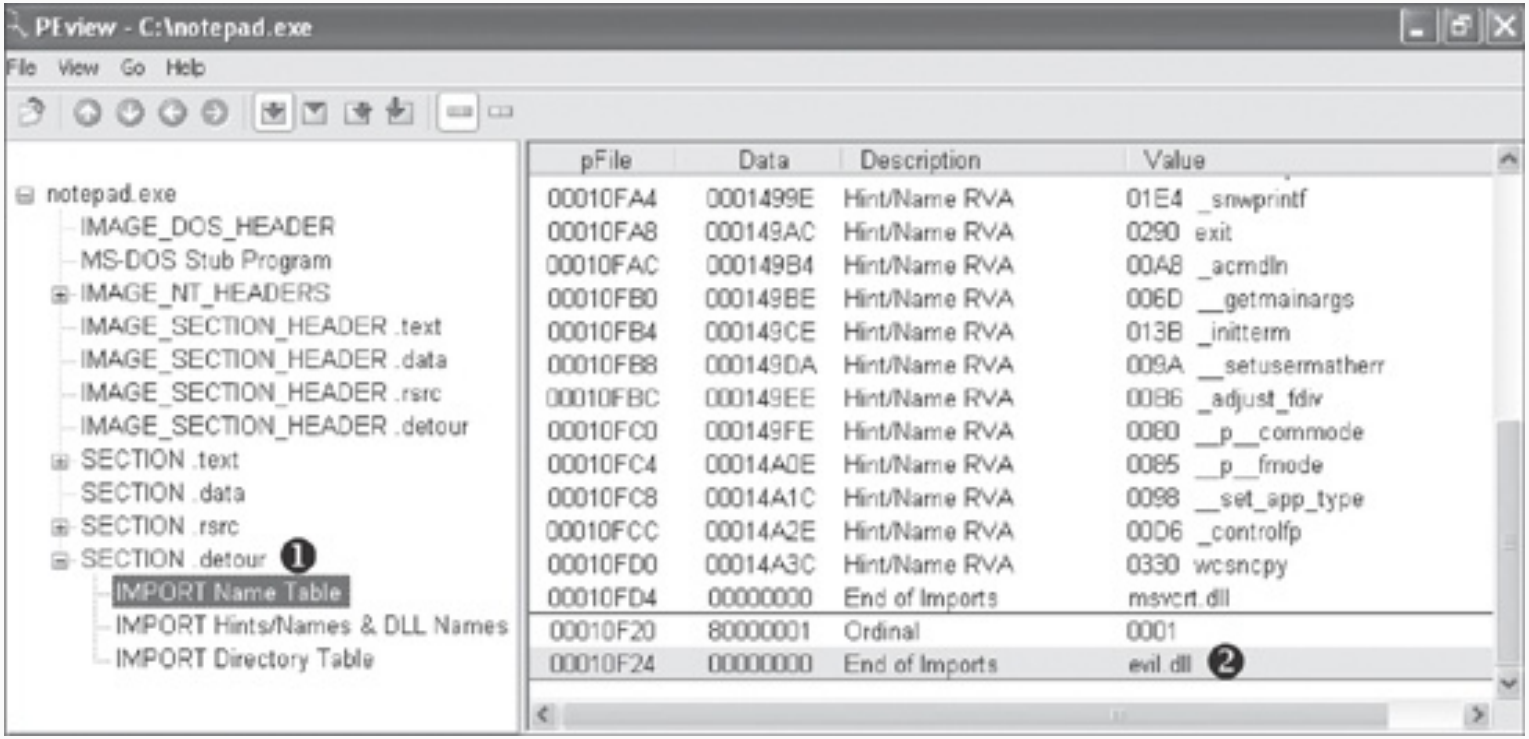
Detours is a library developed by Microsoft Research in 1999. It was originally intended as a way to easily instrument and extend existing OS and application functionality. The Detours library makes it possible for a developer to make application modifications simply.

Malware authors 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.

Detours

A **`PEview`** of Detours used to trojanize **`notepad.exe`** is shown:



Notice in the **`.detour`** section at ❶ that the new import table contains **`evil.dll`**, seen at ❷, which will now be loaded whenever **`Notepad`** is launched, as it continues to operate as usual.

APC Injection

Launcher

Process Injection

Process Hollowing

Hook Injection

Detours

APC Injection

APC Injection

We saw that by creating a thread using `CreateRemoteThread`, we 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)**.

Every thread has a queue of **APC**s attached to it, which are processed when the thread is in an *alertable state*, e.g. when they call `SleepEx`, `WaitForSingleObjectEx`, and `WaitForMultipleObjectsEx`. These functions give the thread a chance to process its waiting APCs.

When the APC queue is complete, the thread continues along its regular execution path.

APC Injection

Malware uses APCs to preempt threads in an alertable state and get executed.

APCs come in two forms:

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

Malware generates user-mode APCs from both kernel and user space using APC injection.

APC Injection: 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`, with these parameters:

- **`pfnAPC`**,
- **`hThread`**, and
- **`dwData`**.

A call to `QueueUserAPC` is a request for the thread whose handle is **`hThread`** to run the function **`pfnAPC`** with the parameter **`dwData`**.

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.

APC Injection: APC Injection from User Space

The following example shows APC injection from a user-mode application, i.e., how malware can use **`QueueUserAPC`** to force a DLL to be loaded in the context of another process:

```
00401DA9        push    [esp+4+dwThreadId]      ; dwThreadId
00401DAD        push    0                       ; bInheritHandle
00401DAF        push    10h                     ; dwDesiredAccess
00401DB1        call    ds:OpenThread
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        ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

APC Injection: APC Injection from User Space

Before we arrive at this code, the malware has already picked a target thread.

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.

APC Injection: APC Injection from User Space

Once a target-thread identifier is obtained, the malware uses it to open a handle to the thread.

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`**.

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: 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`**.

APC Injection: APC Injection from Kernel Space

The following example shows an example of these functions in use in a rootkit:

```
000119BD        push    ebx
000119BE        push    1
000119C0        push    [ebp+arg_4]
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0]
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
```

APC Injection: APC Injection from Kernel Space

The APC must be initialized with a call to **`KeInitializeApc`**. If the 6th parameter (**`NormalRoutine`**) is non-zero, with the 7th parameter (**`ApcMode`**) set to 1, then we are looking at a user-mode type.

`KeInitializeAPC` initializes a **`KAPC`** structure, which must be passed to **`KeInsertQueueApc`** to place the APC object in the target thread's corresponding APC queue. 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`**. Therefore, we would need to look back in the code to check how **`arg_0`** was set to see that **`svchost.exe`**'s threads were targeted.

Next: Data Encoding in Malware