Practical Malware Analysis

Lecture 12 | Covert Malware Launching

Launchers

- Conceal malicious behavior
  - .rsrc
- Require privilege

The main goal of a launcher is to launch or stage the launch of malware in a way that conceals it from a user, a good example of this is a program that extracts the malware from it's own resource section. Launchers often require administrative privileges to run, and therefore may also contain privilege escalation code.

Process Injection

- Popular
- Inject code -> process
    - Stealth
    - Bypass
- *VirtualAllocEx* + *WriteProcessMemory*

Process injection is the most popular means by which malware is launched, by injecting code into an existing process which executes the malicious code. This method will allow code to be run under an existing process (smaller footprint) and may allow the malware to bypass some host-based firewalls or process-specific security mechanisms. Methods involving process injection often contain API calls to *VirtualAllocEx* and *WriteProcessMemory* to allocate space in a remote process and write code to that space.

DLL Injection

1. Find target process
2. Get handle
3. Add resources to memory
4. Create thread

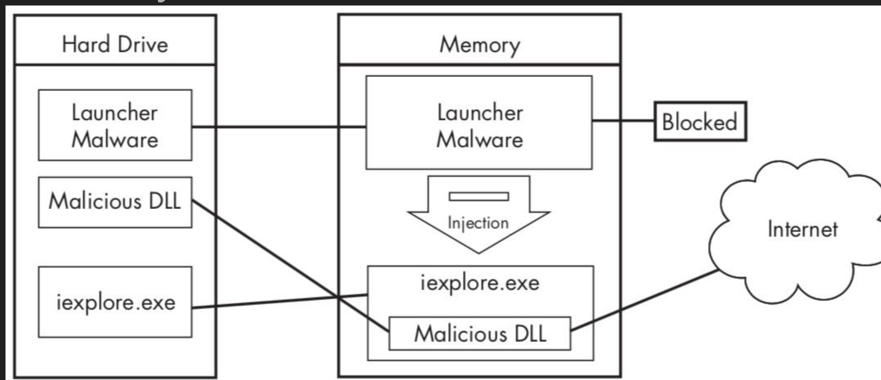

Figure 12-1: DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

DLL injection, the most common form of process injection, works by injecting code into a process that calls the *LoadLibrary* API function to load a DLL in the process' context. The OS then automatically calls the loaded DLL's *DllMain* function.

In the example above, a malicious DLL is injected into Internet Explorer's process space, thereby gaining the same level of access to the Internet as the browser.

Common steps to perform DLL injection are:

1. Call *CreateToolhelp32Snapshot*, *Process32First*, and *Process32Next* to search the process list for the target process
2. Retrieve a handle to the target process with a call to *OpenProcess* using the process' process identifier (PID)
3. Call *CreateRemoteThread* to create a thread for the DLL to run in
    a. Pass in the process handle (*hProcess*)
    b. Pass in the starting point of the injected thread (*lpStartAddress*) and parameter (*lpParameter*), likely *LoadLibrary* + *malicious.dll*
        i. Requires the *LoadLibrary* function to be available to the process, and the name of the malicious dll to exist as a string w/i that memory space as well
        ii. Use *VirtualAllocEx* to allocate space and *WriteProcessMemory* to write the name string to the process' memory space

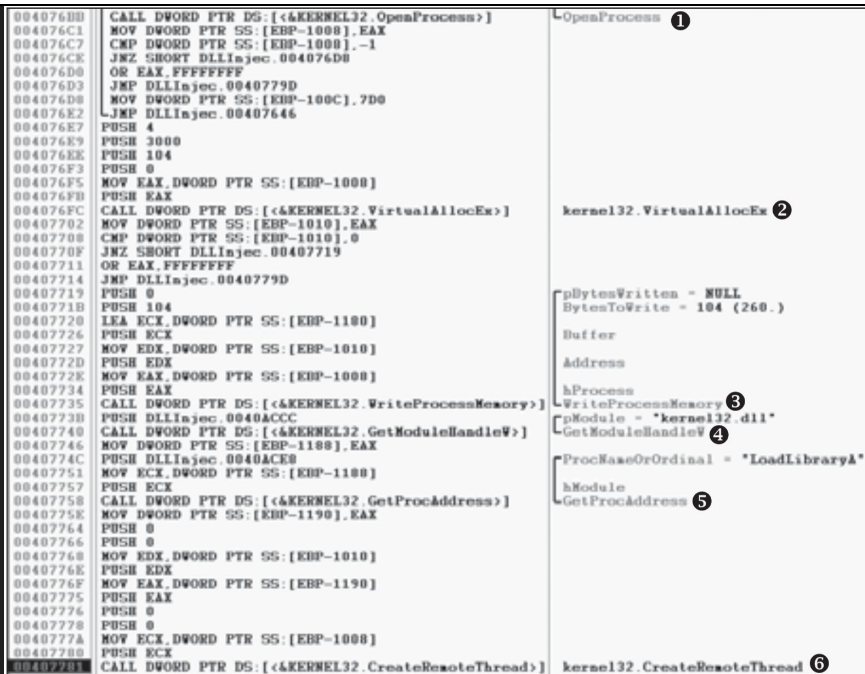

Figure 12-2: DLL injection debugger view

The above illustrates a view of the process outlined on the previous slide, with a view of what the disassembled code would look like in a debugger. One can pick out each API function associated with DLL injection at the numbered locations.

Direct Injection

- Like DLL, but code only
  - Flexible
  - Requires custom code
- Write data
  - *VirtualAllocEx, WriteProcessMemory*
- Write thread code
  - *VirtualAllocEx, WriteProcessMemory, CreateRemoteThread*
    - *lpStartAddress* for remote thread code
    - *lpParameter* for data

Direct injection is the same as DLL injection, but instead of calling LoadLibrary to load a DLL, malicious code (usually shellcode) is injected into the process and ran. While this gives authors more flexibility in what they can run, it requires a lot of customized code to work properly without affecting the host process.

For direct injection, two calls to each of *VirtualAllocEx* and *WriteProcessMemory* will occur to write the data used by the thread, followed by writing the thread code and calling *CreateRemoteThread* with the addresses for the thread code and data as parameters. Consider dumping memory buffers that occur before calls to *WriteProcessMemory* to analyze.

Process Replacement

- Overwrite running process’ memory
  - Disguise
  - Low risk

1. Create suspended process
2. Replace victim process’ memory
3. Point to new entry point
4. Resume suspended

Instead of inserting code into a running process, an author may choose to entirely overwrite that process’ memory space with a malicious executable. This allows malicious code to run, disguised as the victim process (and with its privileges), without the risk of crashing the victim process.

Process Replacement

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPENDED,...);
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

[Picture of sequence of API calls commonly encountered in code performing process replacement]

Hook Injection

- Ensure code runs if <message> occurs
- Ensure <DLL> is loaded

- Local v Remote hooks
  - (Remote) high v low-level

Ex. Keyloggers

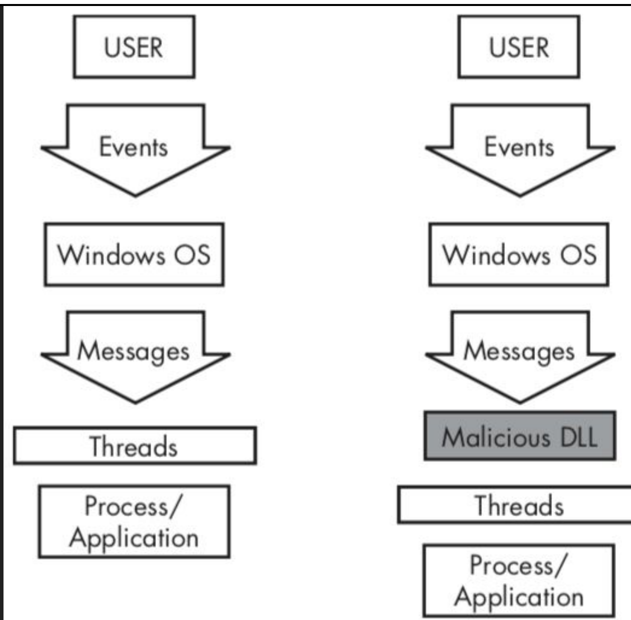

Figure 12-3: Event and message flow in Windows with and without hook injection

Windows hooks are used to intercept messages en route to an application. Malware authors can use hooks to ensure malicious code runs every time a given message occurs, or to ensure a given DLL is loaded in a victim process's memory.

Local hooks may be used to watch (or change) messages intended for an internal process, whereas remote hooks are necessary for messages destined for a remote process (a separate process on the same system). For remote hooks, high-level remote hooks require that the hook procedure is an exported function from a DLL that will be mapped by the OS into the process space of a hooked thread (or all threads). Low-level remote hooks require the hook procedure is within the process that installed the hook, the procedure is notified before the OS processes the event.

A common example of hook injection occurs in keyloggers, where high- or low-level hooks (*WH_KEYBOARD* or *WH_KEYBOARD_LL*) are registered to capture keystrokes. For the high-level hook, the hook will normally run in the context of a remote process, but may also run within the installing process. For the low-level hook, the hook must run in the context of the process that installs it in order to capture keystroke events. Keystrokes may then be logged/altered/etc.

Hook Injection

SetWindowsHookEx parameters

idHook	Type of hook procedure to call
lpfn	Pointer to hook procedure
hMod	Handle to DLL containing **lpfn** procedure (high-level) or to local module where **lpfn** is defined (low-level)
dwThreadId	Thread ID to associate hook with, if 0, hook procedure is associated with all existing threads running in the same desktop as the calling thread (must be 0 for low-level hooks)

**CallNextHookEx*

The primary function used in remote Windows hooking is *SetWindowsHookEx* which takes four parameters described in the table above. The hook procedure may processes incoming messages from the system and do something with them, or it may not. Either way it must call *CallNextHookEx* and pass the message to the next hook procedure in the call chain in order for the system to continue running properly.

Hook Injection

Thread Targeting

1. Load malicious DLL
2. Get address of hook procedure (only calls *CallNextHookEx)*
3. Get thread ID for notepad.exe
4. Hook *WH_CBT* (uncommon)
5. Message occurs -> *DLLMain* runs

```
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

Malware that injects into all threads is likely performing message interception, such as with a keylogger. Malware that only needs to inject a DLL or similar may target a single thread to be less discoverable, and hook an infrequently used Windows message in order to not trigger an IPS.

<picture of assembly listing for hook injection>
Load malicious DLL
Get address of hook procedure (only calls CallNextHookEx)
Get thread ID for notepad.exe
Hook WH_CBT (uncommon)
Message occurs -> DLLMain runs

Detours

Library for extending OS/app functionality

- IAT modification
- Add function hooks to running processes
- Add DLLs to existing binaries
  - .detour section
    - New IAT, modifies PE header w/ *setdll* tool

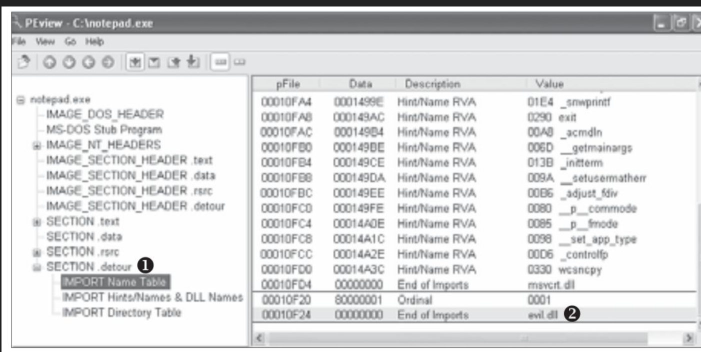

Figure 12-4: A PEview of Detours and the evil.dll

Detours is a library for extending OS and application functionality, added by MSFT in 1999. Malware authors abuse Detours functionality in order to modify the import address table, add function hooks to running processes, and add DLLs to binaries that already exist on disk.

Most commonly, malware modifies the PE structure of a binary, adding a .detour section with the original PE header and a new IAT. The original PE header is then modified to point to the new IAT using the *setdll* tool (part of Detours).

In the example above, *evil.dll* is added to the end of notepad.exe's IAT and will be loaded whenever Notepad is launched.

Asynchronous Procedure Call Injection

Invoke a function on an existing thread

1. *QueueUserAPC*
2. Thread in ‘alterable’ state
    a. *WaitForSingleObjectEx*, *WaitForMultipleObjectsEx*, *SleepEx*
3. Process APCs from APC queue

- Kernel-mode v user-mode

Instead of creating a thread, malware may invoke a function on an existing thread via an asynchronous procedure call (APC). APCs allow a thread to execute some code prior to, or in the midst of, executing its regular code. When a thread is in an alterable state (or prior to running), such as when *WaitForSingleObjectEx* or similar functions are called, a thread can execute any waiting APCs from its queue and then return to its regular execution.

APCs generated for the system or a driver are called *kernel-mode* whereas APCs generated for applications are considered *user-mode*. Malware generates *user-mode* APCs from kernel or user space with APC injection.

The function *QueueUserAPC* accepts a handle to a thread *hThread* to run the function defined by the *pfnAPC* parameter with function parameter provided by *dwData*. Functions commonly called to find the targeted thread include *CreateToolhelp32Snapshot*, *Process32First*, and *Process32Next*, and are often followed with calls to *Thread32First* and *Thread32Next*. Alternatively, malware may call *Nt-* or *ZwQuerySystemInformation* with *SYSTEM_PROCESS_INFORMATION* information class to find a target process.

APC Injection from a user-mode application

```
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
```

Post-target acquisition, the example above opens a handle to the target thread at (1) and then calls *QueueUserAPC* at (2) on the target thread in order to get it to call *LoadLibraryA* on *dbnet.dll*. A popular target for this attack would be svchost.exe, which often has threads in an alterable state. Further, injecting into every thread in svchost.exe will ensure quick execution.

APC Injection from Kernel Space

```
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
```

Get code execution in user space from kernel space with APC injection.

Driver -> dispatch a thread w/ APC (usually shellcode) using *KeInitializeAPC* and *KeInsertQueueAPC*

At (1) the *ApcMode* parameter in combination with a non-zero parameter at (2) indicates user-mode APC injection.
The *KeInitializeApc* function initializes a KAPC struct which is then passed to *KeInsertQueueApc* to place the APC object into the target's APC queue (in this case ESI contains the KAPC struct). At (3) the parameter containing the thread to be injected is pushed, an analyst would have to trace backwards from here to determine how arg_0 was set in order to know the targeted process.