Process Injection Techniques

Part 1 (maybe)

December 20, 2018

*Host’s Note: Much of the content for this Tech Talk was inspired and/or sourced from a July 2017 blog at Elastic (link: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process).
What is Process Injection?

• Process injection is a method of executing code from an executable file in the address space of a separate, live-running target process.

• Why use it?
  • Execution is masked under a legitimate process
    • Exec path points to legit program, (i.e. C:\Windows\system32\svchost.exe)
  • Gain access to a certain process’s memory
  • Gain access to system/network resources that are whitelist restricted
  • Possibility for elevated privileges
  • Some methods provide built-in persistence
Presentation Outline

• This talk focuses on the Windows operating system
• Provides an overview of the following techniques
  • DLL Hijacking
  • Registry Key Modification
  • DLL Injection
  • PE Injection
  • Process Hollowing
  • Hook Injection
  • APC Injection

• All methods take advantage of legitimate OS functionality
• DOES NOT cover Remote Code Execution exploits or shellcode
DLL Hijacking

Abuse the Windows Path Search Order to have your DLL loaded first

AKA Subloading, Sideloadind
Windows Search Order

1) The directory from which the application is loaded
2) The current directory
3) The system directory, usually C:\Windows\System32\ (The GetSystemDirectory function is called to obtain this directory.)
4) The 16-bit system directory - There is no dedicated function to retrieve the path of this directory, but it is searched as well.
6) The directories that are listed in the PATH environment variable.
Advantages

• Provides a built-in persistence mechanism
• Privilege escalation is possible when a search order-vulnerable program is configured to run at a higher privilege level.
• RCE sometimes possible when a program sets its current directory to a remote location (e.g. a web share) before loading a DLL
• Application may attempt to load a DLL that doesn’t exist on a system
  • e.g. when doing some sort of version check
Issues

• If replacing complex DLL, must implement all expected APIs
  • Load the original DLL and divert calls with stub functions
  • Or see PE Export Forwarding
Mitigation: SafeDllSearchMode

- Searches directories with greater restrictions (e.g. %SYSTEMROOT%) before the local directory (e.g. a user's home directory)
  - 1) The directory from which the application is loaded
  - 2) The system directory, usually C:\Windows\System32\ (The GetSystemDirectory function is called to obtain this directory.)
  - 3) The 16-bit system directory - There is no dedicated function to retrieve the path of this directory, but it is searched as well
  - 5) The current directory
  - 6) The directories that are listed in the PATH environment variable
Mitigation: Write Better Code!

- Wherever possible, pass fully qualified path to `LoadLibrary()`, etc.
- Remove the current directory from the standard search path
  - Call `SetDllDirectory()` with an empty string ("")
- Establish a default search path for your process
  - See `SetDefaultDllDirectories()` API
- Create a redirection file for your application
  - Forces the loader to check the application directory first
- Don’t make assumptions about versioning based on DLLs
Registry Key Modification

Modify a specific Registry key to point to your executable file
AppInit_DLLs

- When user32.dll is mapped into a process, and receives a DLL_PROCESS_ATTACH, it will call LoadLibrary on each DLL in the AppInit_DLLs key
  - HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_DLLs
  - HKLM\Software\Wow6432Node\Microsoft\Windows NT \CurrentVersion\Windows\Appinit_DLLs
- Basically, every application that uses GUI items loads user32.dll
- *Technique disabled in Windows 8+ when secure boot is enabled
AppCertDlls

• DLLs are loaded into every process that calls these functions:
  • CreateProcess
  • CreateProcessAsUser
  • CreateProcessWithLogonW
  • CreateProcessWithTokenW
  • WinExec

• AppCertDlls Key:
  • HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls
Image File Execution Options (IFEO)

- Enable a developer to attach a debugger to an application
- When an executable is launched, the debugger present in an application’s IFEO will be attached to it.

IFEO Key:
- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
Advantages

• Often overlooked when searching for autoruns
• Provides a built-in persistence mechanism
• Privilege escalation is possible
Issues

- For Appinit_DLLs
  - Malicious DLL will be loaded by EVERY process that loads user32.dll
  - Technique only works on old versions of Windows
- For AppCertDLLs:
  - Malicious DLL will be loaded by EVERY process that calls the common APIs
- For IFEO:
  - Malicious executable must be able to attach to a running process
Create a thread in the target process that loads a DLL

1) Open the target process
2) Allocate memory in the target process
3) Write the name of a DLL in the allocated memory
4) Call LoadLibrary() in the remote process and pass it the DLL name
5) DllMain is executed when the DLL is loaded
DLL Injection Procedure

• Open the target process with OpenProcess()
  • Make sure to open the process with the required permissions for all steps
• Allocate memory in the target process
• Write the name of a DLL in the allocated memory
• Call LoadLibrary() in the remote process and pass it the DLL name
• DllMain is executed when the DLL is loaded
  • Executed with DLL_PROCESS_ATTACH
Allocating Memory in a Remote Process

- VirtualAllocEx()

```c
LPVOID WINAPI VirtualAllocEx(
    _In_     HANDLE hProcess,
    _In_opt_ LPVOID lpAddress,
    _In_     SIZE_T dwSize,
    _In_     DWORD flAllocationType,
    _In_     DWORD flProtect
);
```

- hProcess requires PROCESS_VM_OPERATION
Writing Memory In A Remote Process

- **WriteProcessMemory()**

```c
BOOL WINAPI WriteProcessMemory(
    _In_    HANDLE  hProcess,
    _In_    LPVOID  lpBaseAddress,
    _In_    LPCVOID lpBuffer,
    _In_    SIZE_T  nSize,
    _Out_   SIZE_T  *lpNumberOfBytesWritten
);
```

- hProcess requires PROCESS_VM_WRITE, PROCESS_VM_OPERATION
- Can write the full path, or use DLL search order we discussed before
Calling LoadLibrary() In The Target Process

- Must first locate the address of the function
- Kernel32.dll is loaded at the same address in every process
  - With ASLR, load address is randomized at startup but is the same in each process
- Call GetProcAddress() from within current process
- Start a new thread in the target process that calls LoadLibrary()

*Alternatively, write a function that performs the LoadLibrary() call*
Starting A Thread In The Target Process

• CreateRemoteThread()

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

• hProcess requires PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, and PROCESS_VM_READ access rights
Code Example

procId = FindProcess(); // Must implement this separately. See CreateTool32HelpSnapshot

hProcess = OpenProcess(<all access rights>, FALSE, procId);
remoteData = VirtualAllocEx(hProcess, NULL, … , MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(hProcess, );

hModule = GetModuleHandle(“kernel32.dll”);
pLoadLibrary = GetProcAddress(hModule, “LoadLibraryW”);

hThread = CreateRemoteThread(hProcess, NULL, 0, pLoadLibrary, remoteData, 0, &threadId);
Issues

• DLL must be written to disk before being loaded into target process
• Limit to what can be accomplished in DllMain
  • DON’T Call LoadLibrary or LoadLibraryEx (either directly or indirectly)
  • DON’T Call CreateProcess. Creating a process can load another DLL
  • Limited to APIs in DLLs that are loaded in every process (i.e. kernel32.dll)
  • CreateThread() is okay as long as you’re not doing any synchronization
• CreateRemoteThread() fails if target process is in a different session
• SeDebugPrivilege required to write memory in a process owned by another user (e.g. Admin -> SYSTEM injection)
Mitigation: Protected Processes

• Introduced in Windows 8.1 (Protected Processes Light, PPL)
  • Extension of system created for DRM in Vista
• Intended to protect anti-malware user-mode services
• Anti-malware vendor must have an ELAM driver installed
• Signing requirements for executable files
  • EXE must be page hash signed
  • Any non-Windows DLLs that get loaded must be signed with the same certs
  • Cert hashes stored in resource file, which is linked to the ELAM driver.
• Still not perfect:
  • [Injecting Code into Windows Protected Processes (James Forshaw)](http://example.com)
Portable Executable Injection

Load full executable into memory of another process

• Default loader functions only work with files on disk
• Re-implement LoadLibrary(), but for a DLL found in memory
In-Memory Loading

- DLL (and loader) written into memory or target process
  - CreateRemoteThread() to call loader function
- Loader will:
  - Find own image's current location in memory
  - Find addresses of LoadLibraryA, GetProcAddress and VirtualAlloc
  - Allocate memory space for new image, load headers and map sections
  - Fix-up IAT and relocation table
  - Call newly loaded image's DllMain with DLL_PROCESS_ATTACH
- Ex: [https://github.com/fancycode/MemoryModule](https://github.com/fancycode/MemoryModule)
- Ex: [https://github.com/stephenfewer/ReflectiveDLLInjection](https://github.com/stephenfewer/ReflectiveDLLInjection)
Advantages

- Malicious code never has to be written to disk
- Not limited to loading DLLs
  - Can really load any PIC code, depending on how complex the task is
Issues

• CreateRemoteThread() fails if target process is in a different session
• SeDebugPrivilege required to write memory in a process owned by another user
• Requires detailed knowledge of PE format
• Also hampered by PPL
Thread Execution Hijacking

Steal a thread of execution from the target process
Suspend, Inject, Resume

- Find the procId and get a threadId (see CreateToolhelp32Snapshot)
- Get a handle to the thread with OpenThread()
- Pause the thread with SuspendThread()
- Write the function and any data with WriteProcessMemory()
  - Can be combined with an in-memory loading technique
- GetThreadContext() to get registers, stack, etc.
- Point execution to newly written function using SetThreadContext()
- Restart suspended thread with ResumeThread()
Modifying Execution in the Target Thread

- **GetThreadContext()**
  - hThread must have THREAD_GET_CONTEXT access right

  ```c
  BOOL GetThreadContext(
      HANDLE hThread,
      LPCONTEXT lpContext
  );
  ```

- **SetThreadContext()**
  - hThread must have THREAD_SET_CONTEXT access right

  ```c
  BOOL SetThreadContext(
      HANDLE hThread,
      const CONTEXT *lpContext
  );
  ```
Advantages

• Malicious code never has to be written to disk
Issues

• SeDebugPrivilege required to write memory in a process owned by another user
• Thread context is architecture dependent
• Very easy to cause a system crash
  • if interrupting NTDLL code, for example
Process Hollowing

Replace the memory space of the target process with a malicious executable
Stealing Execution

- Start process in suspended mode and replace main thread’s code
- Unmap memory of target process with NtUnmapViewOfSection
- Allocate new memory with VirtualAllocEx()
- Write malware using WriteProcessMemory()
  - Can be combined with an in-memory loading technique
- Point execution to newly written code using SetThreadContext()
- Restart suspended thread with ResumeThread()
- Ex: [https://github.com/m0n0ph1/Process-Hollowing](https://github.com/m0n0ph1/Process-Hollowing)
Advantages

• Malicious code never has to be written to disk
• Not as susceptible to crashing as Thread Hijacking
Issues

• Thread context is architecture dependent
Hook Injection

Install a hook routine into the hook chain
These routines are invoked whenever certain events are triggered
• e.g. key presses, mouse clicks
DLL that implements the hook will be automatically loaded in remote process
Hook Injection Procedure

• Insert into the hook chain using SetWindowsHookEx()

```c
HHOOK SetWindowsHookExA(
    Int    idHook,
    HOOKPROC lpfn,
    HINSTANCE hmod,
    DWORD  dwThreadId
);
```

• When the event occurs, the OS checks if the DLL is already loaded
• If not loaded, it will load the DLL and call DllMain
• It will also call the lpfn we passed to SetWindowsHookEx()
• Subsequent events will not reload the DLL
• Uninstalling the hook routine will unmap the DLL
Hook Injection Procedure

• LoadLibrary() to get hModule to your DLL
• GetProcAddress() to get addr to hook function within your DLL
• FindWindow() and GetWindowThreadProcessID() to get threadId (and pid) of desired window
• SetWindowsHookEx() to insert routine in hook chain of desired thread
Hook Injection Procedure

```c
hMod = LoadLibrary(L"C:\path\to\my.dll");

hookProcAddr = GetProcAddress(dll, "myHookFunc");
targetWindow = FindWindow(NULL, L"calc.exe");
getWindowThreadProcessId(targetWindow, &procId);

hHook = SetWindowsHookEx(WH_KEYBOARD, hookProcAddr, hMod, threadId);
```
Issues

• Can only inject into applications that use window messages (i.e. have user32.dll loaded)
Mitigation: User Interface Privilege Isolation

• Prevents lower "integrity level" (IL) processes from sending messages to higher IL processes
• Can be bypassed, but that requires:
  • A valid code signing certificate issued by an approved code signing authority
  • User being attacked must have admin privileges
  • Convincing user to grant use of his/her admin privileges in the UAC prompt.
• Vista+
Asynchronous Procedure Call (APC) Injection

Attach code to the APC Queue of another thread.
Execution upon the thread entering an alterable state.
APCs

- Each thread has a queue of APCs that await execution upon the target thread entering alterable state:
  - SleepEx
  - SignalObjectAndWait
  - MsgWaitForMultipleObjectsEx
  - WaitForMultipleObjectsEx
  - WaitForSingleObjectEx (one of the most commonly called APIs)
- APC generated for the system or a driver is a kernel-mode APC
- APC generated for an application is called a user-mode APC
Using User-Mode APCs

- Get a handle to the desired thread (OpenThread)
  - must have THREAD_SET_CONTEXT access right
- Call QueueUserAPC() with the address of LoadLibrary as the pfnAPC parameter and the DLL path as the data parameter

```c
DWORD QueueUserAPC(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData
);
```
Using Kernel-Mode APCs

- Initialize the APC with
- Pass the APC to KeInsertQueueApc() to place it into the queue of the target thread
Issues

• No guarantee the thread will be alertable. Probably need to queue for all threads.
For Next Time (Maybe)

IAT Hooking
Real world samples
Demos
Questions?