Covert Malware Launching

Covert Malware Launching

- Techniques for executing malware with the goal of avoiding detection
  - Malware wants to hide from task manager, antivirus, etc.

- Analysts must be aware of common covert malware launching techniques
  - Be aware of common Windows API call indicators
  - Find covertly launched malware on a live system

Loader / Launcher

- A category of malware that contains a payload to be covertly executed
- Typically, the payload is obfuscated on disk to hide malicious functionality from static analysis
- Example: A loader contains an obfuscated PE file as a resource. It loads and deobfuscates the resource, then injects into explorer.exe to give it execution.

Process Injection

- Most popular type of covert malware launching
- Malware injects code into a legitimate process
- Typically indicated by VirtualAllocEx and WriteProcessMemory
- 3 major types: DLL injection, direct injection, process replacement

DLL Injection – Part 1

- Type of process injection where a victim process is forced to load a malicious DLL

- Step 1: Loader gets a handle to the victim process
  - Takes a snapshot of the running processes and iterates over them
  - Gets the PID of the victim process
  - Obtains handle to the process using its PID
  - Typical API calls: CreateToolHelp32Snapshot, Process32First, Process32Next, OpenProcess

DLL Injection – Part 2

- Step 2: Loader writes name of DLL in victim process's memory
  - VirtualAllocEx – given handle to victim process, allocate space for the name of the malicious DLL
  - WriteProcessMemory – Writes malicious DLL name to space allocated with VirtualAllocEx

DLL Injection – Part 3

- Step 3: Loader calls CreateRemoteThread with:
  - hProcess = handle to the target process from previous step
  - lpStartAddress = address of LoadLibraryA
  - lpParameter = name of malicious DLL file to be injected

- This forces the target process to create a new thread and call LoadLibrary on the malicious DLL
  - Executes any code in the malicious DLL's DllMain export
  - Why?

Direct Injection – Part 1

- Injects code directly into victim process

- More flexible but has some downsides:
  - Often requires a lot of custom code
  - Might accidentally corrupt the process being injected into

Direct Injection – Part 2

- Step 1: Loader writes any needed data to the victim process
  - Allocate memory for data with VirtualAllocEx
  - Write data to victim process with WriteProcessMemory
- Step 2: Loader writes any needed code to the victim process
  - Allocate memory for code with VirtualAllocEx
  - Write code to victim process with WriteProcessMemory

Direct Injection – Part 3

- Step 3: Loader calls CreateRemoteThread with:
  - hProcess = handle to the target process
  - lpStartAddress = address of injected code
  - lpParameter = address of injected data

- May need to do more stuff
  - For example, resolving any needed imports at runtime with LoadLibraryA / GetProcAddress

Process Replacement – Part 1

- Overwrites the memory space of a running process with a malicious executable
  - Gives the malware the same privileges as the process being replaced

- Step 1: Create malicious process in a suspended state
  - Call CreateProcessA on malicious process
  - Pass parameter dwCreationFlags = 0x4 (CREATE_SUSPENDED)
  - The process's main thread is suspended at the entry point

Process Replacement – Part 2

- Step 2: Get a handle to the victim process
- Step 3: Release memory of a section(s) of the victim process
  - Call ZwUnmapViewOfSection with victim process's handle
- Step 4: Allocate new memory to the victim process
  - Call VirtualAllocEx to allocate memory for new executable

Process Replacement – Part 3

- Step 5: Write malware sections to victim process space
  - Call WriteProcess memory, often in a loop
- Step 6: Modify victim thread's context
  - Call SetThreadContext
  - Change thread's entry point to start of the malicious code
- Step 7: Resume the malicious thread
  - Call ResumeThread

Detection using Process Explorer

- Use the “Verify Process” to check signed files for modifications
- Use the strings view to check for different strings in memory than on disk
- Investigate the lower pane view for unusual DLLs

Hooks

- Users generate events that are sent to the OS
- The OS sends messages created by these events to threads registered to receive them
- A Windows hook can intercept these messages and execute code

Creates a Windows hook

- SetWindowsHookEx
  - idHook – Specifies type of hook to install
  - lpfn – Address of hook procedure
  - hMod – Handle to the DLL containing the hook procedure
  - dwThreadId – Identifier of the thread to hook. If it is 0, it hooks all threads on the system

Hook Injection – Part 1

- Hook injection: method for loading malware using hooks
  - Can run malicious code whenever a particular message is intercepted
  - Can make sure a specific DLL is loaded into a victim process' address space

- Will discuss an example of how hook injection can be used to load a DLL

Hook Injection – Part 2

- Step 1: Loader gets address of hook procedure
  - Loader calls LoadLibraryA on malicious DLL
  - Loader calls GetProcAddress to get the address of the hook procedure in the malicious DLL
- Step 2: Obtain identifier of victim thread
  - Get a handle to the victim thread using CreateToolHelp32Snapshot, Thread32First, Thread32Next, OpenThread
  - Call GetThreadId, passing in the victim thread's handle

Hook Injection – Part 3

- Step 3: Loader calls SetWindowsHookEx with:
  - idHook = type of message to hook (typically an uncommon one)
  - lpfn – Address of malicious DLL's hook procedure
  - hMod – Handle to the malicious DLL
  - dwThreadId = Victim thread's ID

- Step 4: Loader sends a message with same type as idHook parameter to the victim process
  - This forces the victim process to load the malicious DLL, any code in its DllMain is executed