Analysis Report for: LOG_API.TXT

Overall Functionality

The provided code snippet is not C code itself, but rather a log of system calls made by an executable file (`CC455151A5545DD70BC5EAC407B18620.exe`). The log reveals a program that extensively utilizes Windows API functions related to memory allocation, process and system information retrieval, thread manipulation, and library loading/unloading. The large number of `VirtualAllocEx` calls, coupled with `WriteProcessMemory` calls, strongly suggests the executable is performing some form of memory manipulation and potentially code injection. The use of `CreateProcess` hints at the creation of new processes. The presence of `Sleep` calls indicates the program waits at various points during execution. The overall behavior is consistent with malicious activity.

Function Summaries

The log shows calls to the following Windows API functions:

* **`GetModuleHandle(dll)`:** Retrieves a handle to the specified DLL module. Parameters: DLL name (string). Return value: Handle to the module.
* **`VirtualAllocEx(process, type, protection, size)`:** Reserves or commits a region of memory within a specified process. Parameters: Process handle, allocation type (e.g., `MEM_COMMIT`, `MEM_RESERVE`), memory protection (e.g., `PAGE_EXECUTE_READWRITE`, `PAGE_READWRITE`), size of region. Return value: Base address of the allocated region.
* **`FreeLibrary(hModule)`:** Releases the DLL module previously loaded with `LoadLibrary` or `GetModuleHandle`. Parameters: Module handle. Return value: Non-zero on success.
* **`QuerySystemInformation(infoClass)`:** Retrieves system information. Parameters: Information class (integer). Return value: Status code.
* **`QueryProcessInformation(process, infoClass)`:** Retrieves information about a process. Parameters: Process handle, information class (integer). Return value: Status code.
* **`OpenProcessToken(process, desiredAccess)`:** Opens the access token associated with a process. Parameters: Process handle, desired access rights. Return value: Handle to the access token.
* **`OpenProcess(processId, access)`:** Opens a handle to an existing process. Parameters: Process ID, desired access rights. Return value: Process handle.
* **`CreateThread()`:** Creates a new thread within the calling process. Parameters: (various security attributes, start address of thread function, etc). Return value: Handle to the newly created thread.
* **`ResumeThread(hThread)`:** Resumes a suspended thread. Parameters: Thread handle. Return value: Previous suspend count.
* **`IsDebuggerPresent()`:** Checks if a debugger is attached to the process. Return value: Non-zero if a debugger is attached.
* **`CreateEvent(lpName, bManualReset, bInitialState, lpName)`:** Creates a named event. Parameters: Name of the event, manual reset flag, initial state, security attributes. Return value: Event handle.
* **`SystemParametersInfo(uiAction, uiParam, pvParam, fWinIni)`:** Retrieves or sets system-wide parameters. Parameters: Action code, parameter value, pointer to parameter data, flags. Return value: Boolean indicating success or failure.
* **`FindWindow(lpClassName, lpWindowName)`:** Retrieves a handle to a window whose class name and window name match specified strings. Return value: window handle or null if not found
* **`QueryFullProcessImageName(process, dwFlags)`:** Retrieves the full path and file name of the executable file for the given process. Return value: Boolean indicating success or failure.
* **`WriteProcessMemory(process, baseAddress, buffer, size, bytesWritten)`:** Writes data to the memory of a specified process. Parameters: Process handle, base address to write to, data buffer, size of data, number of bytes written. Return value: Boolean indicating success or failure.
* **`GetProcessDEPPolicy()`:** Retrieves the Data Execution Prevention (DEP) policy for a process. Return value: Status code
* **`LdrFindEntryForAddress(address)`:** Retrieves the Load Order Module handle from an address. This is used for locating modules in memory. Return value: handle to the module containing the provided address.
* **`SetTimer(hWnd, nIDEvent, uElapse, lpTimerFunc)`:** Sets a timer. Parameters: window handle, timer ID, timeout in milliseconds, callback function.
* **`GetSystemDefaultLangID()`:** Retrieves the system's default language ID.
* **`TerminateProcess()`:** Terminates a process.
* **`ExitProcess(uExitCode)`:** Terminates the calling process.

Control Flow

The control flow is not directly visible from the system call log. The log only shows the sequence of API calls, not the conditional branching or looping within each function. However, we can infer some aspects:

* The program sequentially makes calls to obtain system and process information.
* The numerous `VirtualAllocEx` calls suggest memory is being dynamically allocated. The order of these calls indicates that the allocated memory may be used in subsequent operations, possibly as buffers for data manipulation or code execution.
* The `WriteProcessMemory` calls directly modify the memory of other processes, highlighting the potential for code injection or other malicious actions.
* `CreateProcess` is used to create new processes, suggesting a possible propagation or execution mechanism.
* `Sleep` calls introduce pauses, possibly to synchronize operations or evade detection.
* `FreeLibrary` is used to unload DLLs, possibly to remove traces of activity or free resources.

Data Structures

No data structures are explicitly visible in the log. However, the program internally manages data structures to handle process handles, memory regions, thread handles, and potentially other internal states.

Malware Family Suggestion

Based on the observed system calls, this executable strongly resembles a **rootkit** or a **malware loader**. The extensive memory allocation, information gathering, and process manipulation strongly suggest attempts to hide itself and potentially inject malicious code into other processes. The `MEM_WRITE_WATCH` allocation flag used with `VirtualAllocEx` is interesting, as this is often used by rootkits to detect attempts to analyze or modify its code in memory. The creation of new processes could point to the capability of the malware to self-propagate or to launch additional payloads. Without the actual code, a more precise classification is not possible, but strong indicators of malicious intent exist. Further analysis using a sandboxed environment and dynamic analysis tools would be required to precisely determine its capabilities and true nature.