Analysis Report for: 935CEE80F3AA1E309823914D3553573B.exe.c

Overall Functionality

This C code appears to be a DLL (Dynamic Link Library) that acts as a wrapper or intermediary for a CAN (Controller Area Network) device driver. It loads multiple CAN driver instances from different DLLs, manages their lifecycle (opening, closing, initialization), and provides a unified interface to access their functionalities (reading/writing data, getting status, etc.). The code also incorporates custom memory management, possibly to avoid using the standard C heap allocation functions. The heavy use of function pointers suggests dynamic loading and dispatching of CAN-related operations. The error handling is rudimentary, primarily setting an error code in `dword_1000BE20`.

Function Summaries

* **`DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)`:** The DLL entry point. Initializes the library on process attach (`fdwReason == 1`) and cleans up on process detach (`fdwReason == 0`). Returns `TRUE` for success, `FALSE` for failure.

* **`sub_10001060(char *Str)`:** Initializes internal data structures based on the path of the DLL. Loads configuration from an INI file ("kerneldll.ini"). Returns `TRUE` for success, `FALSE` otherwise.

* **`sub_100012E0()`:** Cleans up resources, freeing allocated memory and unloading loaded CAN driver DLLs.

* **`sub_100013F0()`:** Reads the number of CAN devices from the configuration INI file. Returns the number of devices.

* **`sub_10001450(unsigned int Value, LPSTR lpReturnedString)`:** Retrieves the path to a specific CAN driver DLL from the INI file based on `Value` (device index). Writes the path to `lpReturnedString`. Returns `0` for success, `1` if the string was found in the file, and sets `dword_1000BE20`.

* **`sub_10001530(int a1)`:** Sets the error code in `dword_1000BE20`. Returns `a1`.

* **`sub_10001540(unsigned int Value)`:** Loads a specific CAN driver DLL and retrieves function pointers to its exported CAN functions. Returns `TRUE` on success, `FALSE` otherwise.

* **`VCI_OpenDevice(unsigned int Value, int a2, int a3)`:** Opens a CAN device. Uses function pointers obtained from `sub_10001540`. Returns the result of the underlying driver's `VCI_OpenDevice` call or an error code.

* **`VCI_CloseDevice(unsigned int a1, int a2)`:** Closes a CAN device. Similar to `VCI_OpenDevice`, using function pointers.

* **`VCI_InitCAN(unsigned int a1, int a2, int a3, int a4)`:** Initializes a CAN channel.

* **`VCI_ReadBoardInfo(unsigned int a1, int a2, int a3)`:** Reads board information.

* **`VCI_ReadErrInfo(unsigned int a1, int a2, int a3, _DWORD *a4)`:** Reads error information.

* **`VCI_ReadCANStatus(unsigned int a1, int a2, int a3, _DWORD *a4)`:** Reads CAN status.

* **`VCI_GetReference(unsigned int a1, int a2, int a3, int a4, int a5)`:** Gets a reference.

* **`VCI_SetReference(unsigned int a1, int a2, int a3, int a4, int a5)`:** Sets a reference.

* **`VCI_GetReceiveNum(unsigned int a1, int a2, int a3)`:** Gets the number of received CAN messages.

* **`VCI_ClearBuffer(unsigned int a1, int a2, int a3)`:** Clears the CAN receive buffer.

* **`VCI_StartCAN(unsigned int a1, int a2, int a3)`:** Starts CAN communication.

* **`VCI_Transmit(unsigned int a1, int a2, int a3, int a4, int a5)`:** Transmits CAN messages.

* **`VCI_Receive(unsigned int a1, int a2, int a3, int a4, int a5, int a6)`:** Receives CAN messages.

* **`sub_10001E27(LPVOID lpMem)`:** Wrapper for memory deallocation (`sub_10002286`).

* **`sub_1000218A(unsigned int a1)`:** Custom memory allocation function.

* **`sub_10002286(LPVOID lpMem)`:** Custom memory deallocation function.

* **`sub_1000253C(FILE *File, char *a2, int a3)`:** Complex function that seems to handle formatted output, potentially to a log file. Its intricate logic involving various data types and conditional branches makes it difficult to fully summarize concisely.

* **`sub_10003787(int a1)`:** Determines the heap type used by the application.

* **`sub_100037B4()`:** Detects the OS version and heap selection method (global or per-process).

* **`sub_100038FC(int a1)`:** Initializes the custom heap manager.

* **`sub_10003959()`:** Destroys the custom heap.

* **`sub_10003A3A(DWORD NumberOfBytesWritten)`:** Handles error reporting, possibly writing to the console or displaying a message box.

* **`sub_10003C1B(_DWORD *a1, int a2)`:** A complex function related to the custom memory manager's free operation.

* **`sub_100043F9()`:** Allocates and initializes the custom heap's data structures.

* **`sub_1000453D(LPVOID *lpMem)`:** Frees a block in the custom heap.

* **`sub_10004593(int a1)`:** Frees multiple blocks from the custom heap.

* **`sub_10004655(unsigned int a1, _DWORD *a2, unsigned int *a3)`:** Finds a block in the custom heap.

* **`sub_100046AC(int a1, int a2, _BYTE *a3)`:** Updates usage counters in the custom heap.

* **`sub_100046F1(unsigned int a1)`:** Allocates a block in the custom heap.

* **`sub_100048F9(int a1, unsigned int a2, unsigned int a3)`:** A helper function for heap allocation, possibly for splitting a block.

* **`sub_1000541D(int a1, int a2)`:** Allocates memory using a custom scheme, potentially utilizing a custom heap.

* **`sub_10006698(unsigned int a1)`:** Maps error codes.

* **`sub_1000696A()`:** Calls a function (`flsall`) which is likely involved in thread-local storage cleanup.

Control Flow

The control flow is largely determined by function calls and conditional checks on function success or error codes (`dword_1000BE20`). The `VCI_*` functions all follow a similar pattern: check for valid device index and whether the device is open; then call the corresponding function from the loaded driver. `sub_1000253C` and the custom memory management functions (`sub_1000218A`, `sub_10002286`, `sub_10004655`, `sub_100046AC`, `sub_100046F1`, `sub_1000453D`, `sub_10004593`, `sub_100048F9`, `sub_1000541D`) have complex control flow due to their role in managing data and memory.

Data Structures

The code uses several arrays and pointers to manage loaded DLL handles, function pointers, and custom heap data structures. The custom heap appears to be implemented as a linked list of free blocks. The structure's details are obscured by the decompiler, but the functions suggest it involves block sizes, pointers to next free blocks and allocated blocks. `dword_1000B984` and `dword_1000BAE8` appear to be lookup tables for error codes.

Malware Family Suggestion

Given its functionality as a CAN bus driver wrapper with custom memory management, and dynamic library loading, this code exhibits characteristics that could be associated with **rootkit** or **driver-based malware**. The custom memory allocation and deallocation routines could be used to hide its presence or operations from standard system tools. The loading of multiple CAN drivers based on a configuration file could enable it to adapt to different environments and evade detection. The obfuscated nature of the decompiled code further suggests an intention for stealth. A thorough analysis of the functions and their behavior with respect to the CAN bus would be needed to make a definitive determination. A malicious version could utilize CAN to perform actions undetected and might be designed to target industrial control systems (ICS) for attacks.