Analysis Report for: E058A94916AF850621202282DFEC2207.c

Overall Functionality

This C code appears to be a highly obfuscated component of malware, likely a rootkit or backdoor. The code heavily relies on Windows API calls and employs extensive memory manipulation, including dynamic memory allocation and deallocation (HeapAlloc, HeapFree, HeapReAlloc), thread management (CreateThread, TerminateThread, GetCurrentThreadId), and potentially hooking (SetWindowsHookExW, UnhookWindowsHookEx). The presence of numerous functions with names like `sub_1800XXXX` indicates obfuscation using a decompiler. The use of SIMD instructions (__m128i) further adds to the complexity and suggests potential data encoding or encryption. The code interacts with console input/output, environment variables, and possibly files, suggesting a capability for command and control or data exfiltration. The numerous weak external references to functions from various DLLs (kernel32, ntdll, user32, etc.) suggest that the malware may attempt to evade detection by loading needed functions at runtime.

Function Summaries

Due to the obfuscation and lack of meaningful names, providing concise summaries for all 657 functions is impractical. However, a representative sample will illustrate the code's nature:

Function Name	Purpose	Parameters	Return Value
`sub_180001040`	Likely creates and returns a NAPI object (Node.js Add-on API) using a helper function.	NAPI environment, pointer to data	NAPI object
`sub_1800010A0`	Creates a NAPI string from a given C string and adds it as a property to a NAPI object.	NAPI environment, C string pointer, error pointer	NAPI object
`sub_180002C50`	Installs and uninstalls keyboard and message hooks, likely for keylogging or monitoring.	Pointer to thread-safe function	Returns a NAPI value, likely success/failure code
`sub_180003450`	Constructs a NAPI Error object, potentially handling exceptions from native code.	NAPI environment pointer, error code	A NAPI error object
`sub_180004824`	Loads a function from a DLL at runtime, likely for dynamic code loading and execution.	Import table (array of function pointers)	Pointer to loaded function
`fn`	The actual Windows hook procedure which processes keyboard events	Hook code, wParam, lParam	LRESULT
`sub_180005664`	Throws a std::system_error exception, likely for error handling in the malware.	Error code	Never returns
`sub_180005BF0`	Likely performs memory deallocation	Memory address, size	N/A
`StartAddress`	Thread entry point for newly created threads.	thread parameters	Never returns

Control Flow (Illustrative Examples)

The control flow in most functions is complex and obfuscated using extensive branching and loops. Analyzing all functions is beyond the scope of this response. Here are a few illustrative examples:

* `sub_180002C50`: This function demonstrates the hook installation and uninstallation. It first checks for an existing thread-local storage value before installing a keyboard hook (`SetWindowsHookExW`) and a message hook. Then it enters a message loop (`GetMessageW`, `TranslateMessage`, `DispatchMessageW`). If it receives a message it processes it then it exits, and finally uninstalls the hooks.

* `sub_180003450`: This functions contains numerous branches for checking exceptions conditions. If an exception is pending (`napi_is_exception_pending`), it retrieves the exception (`napi_get_and_clear_last_exception`). Otherwise it creates a NAPI string from a default error message or from the exception and uses that to create a NAPI Error object.

* `sub_180004824`: This function showcases dynamic function loading. First checks for existing functions and loads one if necessary using `LoadLibraryExA` and `GetProcAddress`. It then calls the imported function. If it encounters an error during function loading it raises an exception.

Data Structures

The code uses several data structures which are difficult to fully understand without access to the original compiler's symbol table information. Many structures' sizes and layouts appear to be determined at runtime. The following is a partial list of data types seen that show important elements:

* `__m128i`: SIMD (Single Instruction, Multiple Data) data type used for vectorized operations. This is often used to obfuscate algorithms and makes static analysis more challenging. It strongly suggests data manipulation.

* Structures starting with `sub_` followed by numerical addresses: Decompiled structures with unclear purpose; Likely represent custom data structures used by the malware.

* Arrays of function pointers: These are frequently used for dynamic code execution, polymorhism and indirect function calls.

Malware Family Suggestion

Given its reliance on hooking, thread manipulation, dynamic DLL loading, memory allocation, and obfuscation techniques, this code is highly suggestive of a **rootkit** or a sophisticated **backdoor**. The keylogging aspect (seen in `sub_180002C50`) further strengthens this assessment. The complex memory operations and SIMD usage are common in advanced malware samples to hinder reverse engineering. More specifically it could also be used as a component in a **rat(Remote access trojan)**. The lack of any standard library function calls beyond Windows APIs and the extensive use of decompiled function names suggests that this is probably extracted from a malware binary.

Disclaimer: This analysis is based solely on the provided decompiled code. A full and accurate assessment would require analyzing the original malware binary in a safe, controlled environment using a combination of static and dynamic analysis tools. The above analysis is not definitive proof of the code being malicious.