Analysis Report for: F3C5ACD71536DAAE6B249A1AAFF27D67.exe.c

Overall Functionality

This C code, likely extracted from a malware sample by a disassembler (Hex-Rays), exhibits characteristics of a sophisticated, polymorphic malware. It performs several operations that obfuscate its true intent and hinder reverse engineering. The code heavily uses cryptographic functions (or functions mimicking them), searches for specific modules and strings within the loaded process's memory (likely to check for presence of security software or specific system files), and appears to dynamically resolve functions. The overall purpose is likely to establish persistence, perform malicious actions (the specific payload is not directly visible), and potentially evade detection. The extensive use of seemingly random function names further suggests an attempt to obfuscate its functionality.

Function Summaries

The code contains a large number of functions (52 in total), many of which are short and seem to be utility or helper functions used in more complex routines. Here are summaries for some of the more important functions:

Function Name	Purpose	Parameters	Return Value
`sub_10001000`	Likely a cryptographic or obfuscation routine.	Integers and a character.	void
`sub_10001190`	Searches for a specific module in the process's loaded modules by name.	Integer (likely a string or wide string), LIST_ENTRY*	Pointer to `_LIST_ENTRY` (module information) or NULL
`sub_100012E0`	Similar to `sub_10001190`, but likely handles wide character strings.	Wide character string, LIST_ENTRY*	Pointer to `_LIST_ENTRY` (module information) or NULL
`sub_100013F0`	Core function that may decrypt and execute a payload; involves string comparison and further module searches	Various integers and pointers, including a string.	Integer or pointer (possibly a handle, or failure indicator)
`sub_100016B0`	This function performs additional operations on a string likely part of decryption or string manipulation for API resolution	String pointer	Integer (handle or status indicator)
`sub_100017F0`	Initializes or sets up important function pointers, likely dynamically resolving some critical functionality.	None	void
`sub_100019F0`	Returns a function pointer, likely dynamically resolved during runtime.	None	Function pointer
`sub_10002000`	Seems to be a key function that processes and possibly decrypts/decodes a data buffer, possibly creating an in-memory structure	Character pointer, unsigned integer (size)	Pointer to unsigned integer (result structure)
`sub_10002530`	Another function that might be involved in payload execution, processes an input string	Integer pointer (data structure), String pointer	Integer (result status)
`sub_100027B0`	Cryptographic function (likely custom encryption/decryption)	Various integers and unsigned integers	Unsigned character
`sub_100028D0`	Another cryptographic or obfuscation routine.	Byte array, Integer (size)	Character
`sub_10002950`	Exception handler (unusual position implies it's a possible anti-debugging measure)	Double pointer to EXCEPTION_RECORD	Status indicator
`sub_100037F0`	Compares two byte arrays, case-insensitively	Two byte arrays, integer (size)	Integer (comparison result)
`sub_10003840`	Similar to `sub_100037F0`, likely performs case-insensitive comparison	Integer, unsigned short pointer	Integer (comparison result)
`sub_10004470`	Main entry point or a major function that orchestrates the actions.	None	Integer (result)
`DllEntryPoint`	Standard DLL entry point function.	HINSTANCE, DWORD, LPVOID	BOOL (success/failure)

Control Flow

The control flow is highly complex and obfuscated, especially within functions like `sub_100013F0`, `sub_100016B0`, `sub_100017F0`, `sub_10002000`, and `sub_10004470`. They involve nested loops, conditional branches based on dynamically calculated values, function pointers, and extensive use of bitwise operations, suggesting sophisticated control flow flattening techniques to hinder static analysis. Many functions contain JUMPOUT statements which refer to unresolved addresses, suggesting that the code is either incomplete, or relies on further runtime modifications.

Data Structures

The code utilizes standard Windows structures like `_LIST_ENTRY` (used for linked lists, likely in traversing loaded modules), `_EXCEPTION_RECORD`, and `_CONTEXT`. The code also uses several unnamed structures (`_UNKNOWN`) and arrays of bytes (`byte_1045xxxx`) which are likely custom data structures used for storing encrypted data, intermediate results, or configuration information. The `_UNKNOWN` type is concerning, showing the compiler couldn't resolve types used.

Malware Family Suggestion

Given the techniques employed, the malware family is difficult to pinpoint precisely without more context (e.g., network behavior, dropped files). However, several indicators suggest a sophisticated, polymorphic file infector or a rootkit-like malware:

* **Polymorphism:** The use of encryption, extensive string manipulation, dynamic function resolution, and custom data structures indicates an attempt to evade signature-based detection.
* **Anti-Debugging/Anti-Analysis:** The exception handler (`sub_10002950`) positioned unusually, the dynamic function resolution and overall

```
obfuscation are strong indicators against debugging and reverse engineering.
* **Module Enumeration:** Searching loaded modules suggests an attempt to identify and potentially disable security software or other competing processes.
* **Persistence:** The code's structure suggests aiming for persistence through methods still not obvious from analysis.

Therefore, classifying this code as a **generic polymorphic malware** with rootkit-like characteristics is the most accurate assessment based solely on its static analysis. Further dynamic analysis is crucial to determine its precise functionality and behavior. The extensive obfuscation clearly points to a determined attempt to conceal its actions.
```