Analysis Report for: golangransomware.txt

Overall Functionality

This C code is a ransomware program. It encrypts files on the victim's system and displays a ransom note in an HTML file. The code uses several functions from external DLLs ("kernel32.dll" and "user32.dll"), suggesting it's designed to run on a Windows system. It incorporates features to:

1. **Obtain System Information:** It retrieves the console window handle and potentially other system details.
2. **Command-line Argument Parsing:** It parses command-line arguments using `flag__ptr_FlagSet_String` and `flag__ptr_FlagSet_Parse` functions, likely for options like file paths or ransom amounts.
3. **Random String Generation:** It generates a random string for encryption purposes using `example_com_m_load_RandomString`.
4. **File Encryption:** It iterates through specified folders (hardcoded and potentially user-specified), checks file extensions, and encrypts files using `example_com_m_load_Encrypt` or a similar function. RSA encryption is heavily hinted at by the presence of a public key.
5. **Ransom Note Generation:** It constructs a ransom note as an HTML file, including the ransom amount (potentially customizable), contact information (hardcoded and potentially customizable), and instructions.
6. **Ransom Note Placement:** The ransom note is written to a file named "Encrypt.html".
7. **Log Clearing (Potentially):** It calls `example_com_m_load_Clearlog` at the end which implies attempts to clean up any logs the malware may have generated.

Function Summaries

* **`main_main()`:** This is the main function of the program. It orchestrates all the other functions to achieve the ransomware's goal. It takes no parameters and returns void.

* **`runtime_newobject(...)`:** Allocates memory for a new object of a specified type. Parameters include a type descriptor. Returns a pointer to the newly allocated object.

* **`runtime_gcWriteBarrierCX(...)` / `runtime_gcWriteBarrierDX(...)`:** These are garbage collection functions. Their exact function depends on the garbage collection implementation used by the runtime environment.

* **`syscall__ptr_LazyProc_Call(...)`:** Calls a function from a dynamically loaded library using the `syscall_LazyProc` structure. Parameters are a pointer to the syscall_LazyProc structure (containing function pointer and library), and the function arguments. Return value depends on the underlying called function.

* **`flag__ptr_FlagSet_String(...)`:** This function adds a string flag to a flag set. Parameters include the flag set, the flag name (e.g., "p", "m", "i", "e"), the flag length, the flag's type, and its associated string value. It returns a pointer to the flag in the set.

* **`flag__ptr_FlagSet_Parse(...)`:** Parses a flag set. It takes the flag set and parsed options. Return value is not explicitly stated but likely relates to a status.

* **`runtime_panicSliceB(...)`:** Likely a function that handles errors and crashes the program. It takes error details.

* **`example_com_m_load_RandomString(...)`:** Generates a random string of a specified length. Parameters include the length, and other values for random seed calculation. Returns the generated string.

* **`example_com_m_load_GetDriveLetters()`:** This function retrieves a list of drive letters on the system. The return value is likely a data structure containing this information.

* **`example_com_m_load_EncryptWithRSA(...)`:** Encrypts data using RSA encryption. Parameters include the data to encrypt, the public key, and potentially other parameters. The return value is the encrypted data.

* **`runtime_concatstrings(...)`:** Concatenates multiple strings. Parameters include pointers to the strings and their lengths. Returns the concatenated string.

* **`syscall_Getwd(...)`:** This function retrieves the current working directory. Parameters include a buffer to store the path. Return value is the path or an error code.

* **`loc_461286(...)`:** This is a placeholder name for a function; the address `loc_461286` suggests that this function is potentially related to string or array manipulation, possibly for handling the ransom note parameters.

* **`example_com_m_load_TraverseFolder(...)`:** Traverses a folder recursively, likely to identify files for encryption. Parameters include the path to the folder, and likely control options. Returns likely a list of files and directories.

* **`example_com_m_load_CreateHTMLFile(...)`:** Creates a file and writes the content of a HTML file. Parameters include the path and the HTML content. Returns likely the file handle or status.

* **`example_com_m_load_CheckFileExtension(...)`:** Checks if a file has a certain extension. Parameters include the filename, and likely the extension to check. Returns a boolean value (true if it matches, false otherwise).

* **`example_com_m_load_Encrypt(...)`:** Encrypts a file. Parameters include the file path, potentially encryption key, etc. The return value likely indicates success or failure.

* **`example_com_m_load_EncryptStart(...)`:** A function seemingly to encrypt files before the main loop if the `v118` condition is false.

* **`example_com_m_load_Clearlog(...)`:** Likely a function intended to erase logs or other traces left by the ransomware.

Control Flow

The `main_main()` function's control flow is largely sequential, with branching primarily based on:

* **Console Window Existence:** It checks if a console window exists (`v7`). If it does, it shows it using a `ShowWindow` call.
* **Command-line Arguments:** The program heavily relies on parsing command-line flags. The success of parsing and the presence of specific flags determine the behavior.
* **File Traversal:** A loop iterates through folders identified for encryption; encryption within the loop is conditioned on the file extension.

Data Structures

The code utilizes several custom data structures, notably:

* **`syscall_LazyDLL`:** This struct likely holds information for dynamically loading DLLs, including the DLL's name and a handle.
* **`syscall_LazyProc`:** This struct points to a function inside a dynamically loaded DLL, used for calling API functions.
* **`_2_uintptr`:** An array of two unsigned integer pointers. It's used to pass arguments to the `ShowWindow` function.
* **Flag Sets:** The code uses a `FlagSet` data structure (pointed to by `qword_610D08`) to manage command-line arguments.

Malware Family Suggestion

Based on its functionality, this code strongly resembles a **ransomware** program, specifically a type that encrypts files and displays a ransom note demanding payment in cryptocurrency (USDT). The use of RSA encryption, the ransom note's contents, and the targeting of system files all point towards this classification. The HTML ransom note itself is quite sophisticated and includes CSS styling. The use of a public key suggests an asymmetric encryption scheme was used, making decryption significantly more difficult without access to the private key.

Further Analysis Notes

The code's heavy use of obfuscation techniques (custom data structures, function names like `example_com_m_load_*`, unclear runtime environment) makes a complete analysis difficult without further reverse-engineering. The actual encryption algorithm and key management are hidden within external functions, requiring deeper investigation to fully understand. The presence of potential garbage collection hints that it may be running in a managed environment (such as Go or another managed language compiled to C). The large, embedded public key is also noteworthy, as it makes the code less portable but increases the likelihood of identifying the malware family or authors through analysis of the key itself.