Analysis Report for: project.out.txt

Overall Functionality

This C# code implements a program that appears to decode and process encrypted data, ultimately leading to the execution of some external code. It uses multiple encoding schemes (Ascii85 and a custom "Flutter" encoding), RC4 encryption, and SHA256 hashing. The core logic involves decoding several Base85 and Flutter encoded strings, combining them, decrypting a portion using RC4, loading the resulting bytes as an assembly, and finally invoking a method within that assembly. This strongly suggests malicious intent.

Function Summaries

* **`Ascii85.DecodeBlock()`**: This private method is part of the Ascii85 decoding process. It has no parameters and is overloaded; it calls another `DecodeBlock` method to handle a specific number of bytes.

* **`Ascii85.DecodeBlock(int bytes)`**: This private method decodes a block of Ascii85 encoded data into a byte array. It takes the number of bytes to decode (`bytes`) as a parameter. It doesn't return a value; it modifies the internal `_decodedBlock` byte array.

* **`Ascii85.Decode(string s)`**: This public method decodes a complete Ascii85 encoded string into a byte array. It takes the Ascii85 encoded string (`s`) as input and returns the decoded byte array. It handles the special 'z' case in Ascii85 encoding and throws exceptions for invalid characters.

* **`CodeGen.Ascii85ToBytes(string data)`**: This static method is a wrapper around `Ascii85.Decode`. It takes an Ascii85 encoded string (`data`) and returns the decoded byte array.

* **`CodeGen.FlutterToBytes(string data)`**: This static method decodes a "Flutter" encoded string into a byte array. It uses a custom decoding scheme defined within the `Flutter` class.

* **`CodeGen.GetBytes()`**: This static method orchestrates the decoding of multiple Ascii85 and Flutter encoded strings. It combines the decoded byte arrays from these strings into a single `MemoryStream`. It returns the combined byte array from the `MemoryStream`.

* **`Flutter.Decode(string s)`**: This public method decodes a complete "Flutter" encoded string. It uses a custom decoding algorithm, handling different prefixes (' ', '.', '_') to distinguish various parts of the encoded data. The method throws exceptions for invalid input.

* **`Form1.Form1()`**: Constructor for the `Form1` class (a Windows Form). It hides and minimizes the form, setting its opacity to 0.

* **`Form1.getPharse()`**: Returns the internal `pharse` byte array.

* **`Form1.WndProc(ref Message msg)`**: Overrides the `WndProc` method to handle Windows messages. It specifically intercepts message 1032 (`WM_USER + 1032 - 1024`), copying data from the message's `LParam` to the `pharse` array and then closing the form.

* **`Program.CalcHashThread(object ctx)`**: This method is executed in a separate thread. It takes a handle to a window (`intPtr`) as input. It computes a SHA256 hash based on `Program.seed`, iteratively modifying it until a specific condition is met, and then sends a message (1032) to the specified window containing a new hash.

* **`Program.SendMessage(IntPtr hwnd, int msg, IntPtr wParam, IntPtr lParam)`**: A Windows API call to send a message to a window.

* **`Program.DecodeVarint32(byte[] buffer, int offset, ref uint value)`**: Decodes a variable-length integer from a byte array. It takes the byte array (`buffer`), offset (`offset`), and a reference to a uint variable (`value`) to store the decoded integer. It returns the number of bytes read.

* **`Program.ConvertHexStringToByteArray(string hexString)`**: Converts a hexadecimal string to a byte array. It takes a hexadecimal string (`hexString`) as input and returns the equivalent byte array.

* **`Program.Main(string[] Args)`**: The main entry point of the program. It initializes a form, creates a thread for hash calculation, decodes a data blob, performs RC4 decryption, loads and executes an embedded assembly. This is the core logic of the program.

* **`RC4.Apply(byte[] data, byte[] key)`**: Implements the RC4 stream cipher. It takes the data to be encrypted/decrypted (`data`) and the key (`key`) as input and returns the resulting byte array.

Control Flow

* **`Ascii85.Decode(string s)`**: This function iterates through the input string. For each character, it checks if it's valid (within the Ascii85 range) or a whitespace/control character (which is ignored). If 'z' is encountered and the current block is empty, it appends four null bytes; otherwise, it throws an exception. Valid characters are used to construct a uint value, which is then decoded into bytes using `DecodeBlock` when a full 5-byte block is ready. The last block is handled separately to avoid single byte errors.

* **`CodeGen.GetBytes()`**: This function sequentially calls `Ascii85ToBytes` and `FlutterToBytes` on hardcoded strings, writing the resulting byte arrays into a `MemoryStream`. The order and the contents of the strings are crucial. This suggests the combined data represents an encrypted or encoded payload.

* **`Flutter.Decode(string s)`**: The function iterates through the input string. It distinguishes between three prefixes (' ', '.', '_') to determine how to

decode subsequent characters. Invalid prefixes or characters cause exceptions to be thrown. This custom encoding likely obfuscates the original data.

* **`Program.Main(string[] Args)`**: This function first checks for command-line arguments ("-b" followed by a hex string). If present, it assumes the provided hex string contains the decrypted key and proceeds directly to loading and executing the embedded assembly (after combining the decrypted key with the initial 32 bytes of data). Otherwise, it spawns `CalcHashThread` which computes a hash and sends a message to the hidden `Form1` instance. `Form1` receives this message (via its overridden `WndProc`), copies it into its `pharse` array, and closes. Then, the main thread checks if a specific decryption condition (`array5[0] == 77 && array5[1] == 90`) is met before loading and executing the embedded assembly (using reflection) with the initial data and the decoded payload as parameters. This demonstrates a stage based execution.

Data Structures

* **`Ascii85._encodedBlock`**: A 5-byte array used as a buffer during Ascii85 decoding.
* **`Ascii85._decodedBlock`**: A 4-byte array used as a buffer during Ascii85 encoding.
* **`Ascii85._tuple`**: A uint variable that accumulates the decoded Ascii85 value before being converted into bytes.
* **`Ascii85.pow85`**: An array of uint values representing powers of 85, used in the Ascii85 decoding algorithm.
* **`CodeGen.GetBytes()` `MemoryStream`**: Used to aggregate decoded bytes from various sources.
* **`Form1.pharse`**: A 32-byte array to store data received from the `CalcHashThread`.
* **`Program.seed`**: A 16-byte array; its value is derived from the output of `CodeGen.GetBytes()`. This acts as a seed for the SHA256 hashing process.
* **`Program.RedistData`**: A large byte array (likely containing redistributable data). It is passed to the invoked method in the embedded assembly.
* **`Program.dictionary`**: A `Dictionary` storing decoded blocks of data identified by their index. It likely represents various components of the malware payload.
* **`RC4`'s internal `array`**: A 256-element integer array used in the RC4 algorithm's key scheduling and encryption/decryption process.
* **`RC4`'s internal `array2`**: A 256-element integer array used as a key schedule for the RC4 algorithm

Malware Family Suggestion

Given the code's functionality, it exhibits characteristics consistent with a **downloader/dropper** type of malware. The stages of operation—decoding multiple encoded strings, RC4 decryption of a key, loading and executing an embedded assembly using reflection—strongly points to this type of malware. The embedded assembly is the final payload, likely a more sophisticated piece of malware that performs the actual malicious actions. The command-line argument processing suggests it may also be capable of receiving and executing additional payloads delivered via command line arguments. The use of obfuscation techniques like Ascii85 and custom encoding enhances its stealth and makes analysis more difficult. It also shows characteristics of a polymorphic malware because its encryption and encoding would be different each time it runs.