Analysis Report for: Sample2.cs

Overall Functionality

This C# code implements a custom loader for a .NET assembly. It's structured in several classes: `Form1`, `Program`, `Ascii85`, and `Flutter`. The `Program` class is the entry point, containing the main execution logic. It receives a hex-encoded string as a command-line argument, decodes it, and performs several steps involving hashing, decryption (using RC4), and loading and executing a .NET assembly. The `Ascii85` and `Flutter` classes handle custom encoding and decoding schemes used to obfuscate the data within the hex string. The `Form1` class acts as a minimal Windows Forms application that receives the decrypted key and assembly data through inter-process communication. The final result is the execution of a hidden .NET assembly.

Function Summaries

* **`Form1::Form1()`**: Constructor for the `Form1` class. Initializes the component, hides the form, and sets its opacity to 0. It's designed to be invisible to the user.
* **`Form1::getPharse()`**: Getter for the `pharse` byte array, which will hold the decrypted key. Returns a byte array of length 32.
* **`Form1::WndProc(ref Message msg)`**: Overrides the `WndProc` method to handle Windows messages. Specifically, it intercepts message 1032 (`WM_USER + 1032 - 1024`), copies the data from the message's `LParam` (a pointer to a byte array) into the `pharse` array, and closes the form.
* **`Program::SendMessage(IntPtr hwnd, int msg, IntPtr wParam, IntPtr lParam)`**: A P/Invoke method that sends a Windows message to a specified window handle. Not directly part of the obfuscation/loading process but crucial for inter-process communication.
* **`Program::CalcHashThread(object ctx)`**: A thread function that calculates a SHA256 hash. It takes a window handle (`IntPtr`) as context. It iteratively calculates hashes until a specific condition is met (three leading zero bytes), generating a key. The resulting key and other data are then sent to the main thread using `SendMessage`.
* **`Program::DecodeVarint32(byte[] buffer, int offset, ref uint value)`**: Decodes a variable-length integer (varint) from a byte array. Returns the number of bytes read from the buffer. Used to parse lengths of embedded data within the main data blob.
* **`Program::ConvertHexStringToByteArray(string hexString)`**: Converts a hexadecimal string to a byte array. Used to decode the command line argument.
* **`Program::Main(string[] Args)`**: The main entry point of the program. It orchestrates the entire loading and execution process: decodes data, starts a thread for key generation, performs RC4 decryption, loads the assembly, and invokes the `Draw` method of the loaded assembly.
* **`CodeGen::Ascii85ToBytes(string data)`**: Decodes a string using the Ascii85 encoding. Returns a byte array.
* **`CodeGen::FlutterToBytes(string data)`**: Decodes a string using the custom `Flutter` encoding scheme. Returns a byte array.
* **`CodeGen::GetBytes()`**: This function concatenates multiple Ascii85 and Flutter encoded strings into a single byte stream. The encoded strings are hardcoded within the function. It is the source of the data that will be processed.
* **`Flutter::Decode(string s)`**: Decodes a string using the Flutter encoding scheme. The decoding logic is implemented using a state machine that distinguishes between different prefixes (' ', '.', '_'). Returns a byte array.
* **`RC4::Apply(byte[] data, byte[] key)`**: Implements the RC4 stream cipher to decrypt data. Takes data and a key as input and returns the decrypted byte array.

Control Flow

* **`Program::Main(string[] Args)`**:
1. Creates a `Form1` instance.
2. Creates a dictionary to store decoded data blocks.
3. Creates and starts a thread (`CalcHashThread`) to calculate a hash-based key.
4. Calls `CodeGen::GetBytes()` to obtain the initial encoded data blob.
5. Checks if the first two bytes of the data are 'KS'. If so, parses the data blob according to its structure (varints indicating block sizes, data blocks themselves, and the seed).
6. If the command-line argument provides a key, it uses it directly. Otherwise, it waits for the `CalcHashThread` to finish and send the key via `SendMessage`.
7. If decryption successful and the first two bytes of the decrypted data are 'MZ', it loads and executes the decrypted assembly.
* **`Program::CalcHashThread(object ctx)`**:
1. Receives the window handle.
2. Calculates an initial SHA256 hash of the `seed`.
3. Iterates, incrementing a counter, and modifying a portion of the hash until the first three bytes of the resulting hash are zero. This creates the decryption key.
4. Packages the seed, the generated key, and the string ".NET4" into a byte stream.
5. Computes the SHA256 hash of the byte stream and allocates memory to store it.
6. Copies the hash into the allocated memory and sends it to `Form1` using `SendMessage`.
* **`Flutter::Decode(string s)`**:
1. Iterates through the input string.
2. Uses a state variable (`c`) to track the current prefix (initially '@').
3. A flag (`flag`) indicates whether a prefix is expected.
4. Based on the character and the current state, it either throws an exception (invalid character), outputs a byte (decoded character), or updates the state to expect a following character.

Data Structures

* **`Form1::pharse`**: A byte array of size 32. This array is used to store the decrypted key received from the `Program` class through inter-process

communication.
* **`Program::RedistData`**: A large byte array (presumably containing a redistribution data). This is hardcoded and is used during the execution of the loaded assembly.
* **`Program::seed`**: A byte array of size 16. This array holds the seed data used to generate the decryption key.
* **`Program::dictionary`**: A `Dictionary`. This dictionary stores decoded blocks from the input data. The key represents an index of the data block. The values are byte arrays.
* **`Ascii85::_encodedBlock`**: A byte array of size 5 used for internal operations in the Ascii85 decoding.
* **`Ascii85::_decodedBlock`**: A byte array of size 4 used for internal operations in the Ascii85 decoding.
* **`Ascii85::_tuple`**: A uint variable used to accumulate values during Ascii85 decoding.
* **`Ascii85::pow85`**: A uint array containing powers of 85, used in Ascii85 decoding.
* **`RC4::array`**: An integer array of size 256 used for the RC4 state.
* **`RC4::array2`**: An integer array of size 256 storing the key schedule for RC4.
* **`RC4::array3`**: A byte array used for storing the result of the RC4 encryption/decryption.

Malware Family Suggestion

Given the code's functionality—loading and executing a hidden .NET assembly after a multi-stage decryption process using custom encoding schemes and a hash-based key generation—this strongly resembles a **dropper** or a **downloader**. Droppers are often used to deliver a payload (the hidden .NET assembly) onto a compromised system. The complexity of encoding, hashing and decryption aims at making reverse engineering and analysis more difficult, a characteristic common in many malware families. The fact that it utilizes the RC4 cipher and SHA256 hashing is not uncommon in malware. The final determination of the exact malware family would require analysis of the hidden .NET assembly itself. Without knowing the contents of the loaded assembly, it is impossible to provide a more specific malware family classification. However, the loader itself shows techniques consistent with obfuscation and anti-analysis strategies used by various malware families.