Analysis Report for: C61D70AA2F9D37F6B7A340225DA18103.cs

Overall Functionality

This C# code is a sophisticated obfuscation layer designed to hide the functionality of a payload. It's not directly malicious itself, but serves as a protector for malware. The code decrypts and decompresses embedded LZMA-compressed data (likely containing a .NET assembly), loads that assembly, and then executes a specific method within it. The `Resolve` function further obfuscates the loading of dependent assemblies by encrypting assembly names and loading them from embedded resources. The entire process is heavily reliant on cryptographic techniques and custom data structures, making reverse engineering considerably difficult.

Function Summaries

* **`Decrypt(uint[] A_0, uint A_1)`:** This function decrypts an array of unsigned integers (`A_0`) using a key derived from `A_1`. It employs a custom, seemingly random, series of bitwise XORs, additions, and multiplications. The result is a byte array, which is then decompressed. It returns a `GCHandle` to the decrypted and decompressed byte array (pinned in memory).

* **`Main(string[] A_0)`:** This is the entry point of the program. It decrypts and decompresses a large array of unsigned integers (the embedded LZMA-compressed payload), loads the resulting assembly using `Assembly.Load`, and executes a specified method within that assembly, passing command-line arguments (`A_0`) if available. It returns an integer value obtained from the invoked method.

* **`Resolve(object A_0, ResolveEventArgs A_1)`:** This function is an event handler for `AppDomain.CurrentDomain.AssemblyResolve`. It's triggered when the runtime needs to load an assembly that it cannot find. This function decrypts the assembly name (`A_1.Name`), retrieves the corresponding LZMA compressed assembly from the embedded resources using the decrypted name as the resource key (Base64 encoded), decrypts and decompresses it, and loads it using `Assembly.Load`. It returns the loaded `Assembly` or `null` if loading fails.

* **`Decompress(byte[] A_0)`:** This function decompresses a byte array (`A_0`) using LZMA decompression. It reads the LZMA properties from the beginning of the array, reads the uncompressed size, performs the decompression, and returns the decompressed byte array.

Control Flow

* **`Decrypt`:** The function's control flow is straightforward. It first generates two arrays of pseudo-random numbers based on the input key. Then, it iterates through the input array (`A_0`), XORing each element with a corresponding value from the pseudo-random arrays, performing additional arithmetic operations and writing the results to a new byte array. Finally, the byte array is passed to the `Decompress` function before being allocated in memory and further XORed with a value from another pseudorandom number sequence.

* **`Main`:** The `Main` function has a linear control flow. It first decrypts a large hardcoded array. The resulting byte array is loaded as a module. Then it resolves a method from this module using a key derived from the manifest module and invokes it with the command line arguments. Error handling is minimal; the program returns 0 if the invoked method doesn't return an integer.

* **`Resolve`:** The `Resolve` function follows a conditional path. It first checks if the assembly name is sufficiently short and then encrypts it. Then it attempts to load the resource based on the encrypted name. If the resource is found, it decrypts and decompresses it, loads it, and returns the loaded assembly. Otherwise, it returns `null`, indicating failure to load the requested assembly.

* **`Decompress`:** The `Decompress` function's control flow involves reading the LZMA properties and uncompressed size from the input byte array. Then it uses LZMA decompression to decompress the rest of the array and returns the decompressed data.

Data Structures

* **`uint[]`:** Used extensively to store encrypted data (both input and intermediate results) and for the embedded LZMA compressed payload.

* **`byte[]`:** Used for storing decrypted data, LZMA compressed and decompressed data, and the decryption key (`key`).

* **`.DataType`:** A large, suspiciously sized struct (20048 bytes). It likely contains further padding or encrypted data that is not directly accessed but which contributes to the size and obfuscation of the structure. It is marked with `hasfieldrva` suggesting the runtime data location is linked to the runtime image itself.

* **`.BitDecoder`:** A structure used in LZMA decompression for probabilistic bit decoding.

* **`.BitTreeDecoder`:** A structure used in LZMA decompression for decoding from a bit tree.

* **`.Decoder`:** A class handling the bit stream reading and normalization during LZMA decompression.

* **`.LzmaDecoder`:** A class implementing the LZMA decompression algorithm.

* **`.OutWindow`:** A class managing the output buffer and stream writing during LZMA decompression.

* **`.State`:** A structure representing the internal state machine used by the LZMA decoder.

Malware Family Suggestion

Given the heavy use of encryption, LZMA compression, and the dynamic loading of a hidden assembly, this code strongly suggests a **packer/protector** used by various malware families. The code itself isn't a specific malware family, but rather a tool to make other malware harder to analyze. The techniques used are common in advanced malware samples aiming to evade detection and analysis. The specific malware family would depend on the contents of the decrypted and loaded assembly. Without that information, a precise malware family cannot be definitively determined.