Analysis Report for: 6D2B9B44A58D373CAAA07740D5DE95D8.cs

Overall Functionality

This C# code implements a library for JSON Web Encryption (JWE) and JSON Web Signature (JWS) operations. It supports various encryption and key management algorithms, including AES-CBC-HMAC, AES-GCM, and ECDH-ES, along with RSA and ECDSA signatures. The code heavily utilizes cryptographic primitives from the .NET framework and also interacts with the Windows CNG (Cryptography Next Generation) API via P/Invoke. Notably, the `Qgvkzq` class in the `Qaoeely` namespace exhibits characteristics highly suspicious of malicious intent, downloading and executing code from a remote server.

Function Summaries

* **`AesCbcHmacEncryption` class:** Implements JWE encryption and decryption using AES-CBC and HMAC-SHA.
* `Encrypt(byte[] aad, byte[] plainText, byte[] cek)`: Encrypts plaintext using AES-CBC and generates an authentication tag using HMAC. Returns IV, ciphertext, and authentication tag.
* `Decrypt(byte[] aad, byte[] cek, byte[] iv, byte[] cipherText, byte[] authTag)`: Decrypts ciphertext using AES-CBC and verifies the authentication tag. Returns the plaintext.
* `KeySize`: Returns the key length in bits.
* `ComputeAuthTag(byte[] aad, byte[] iv, byte[] cipherText, byte[] hmacKey)`: Computes the HMAC authentication tag.

* **`AesGcm` class:** Implements JWE encryption and decryption using AES-GCM, using the Windows BCrypt API for crypto operations.
* `Encrypt(byte[] key, byte[] iv, byte[] aad, byte[] plainText)`: Encrypts plaintext using AES-GCM with BCrypt. Returns ciphertext and authentication tag.
* `Decrypt(byte[] key, byte[] iv, byte[] aad, byte[] cipherText, byte[] authTag)`: Decrypts ciphertext using AES-GCM with BCrypt and verifies authentication tag. Returns plaintext.
* `MaxAuthTagSize(IntPtr hAlg)`: Gets the maximum size of the authentication tag.
* `OpenAlgorithmProvider(string alg, string provider, string chainingMode)`: Opens a BCrypt algorithm provider.
* `ImportKey(IntPtr hAlg, byte[] key, out IntPtr hKey)`: Imports the encryption key into BCrypt.
* `GetProperty(IntPtr hAlg, string name)`: Retrieves a BCrypt algorithm property.

* **`AesGcmEncryption` class:** Implements a JWE algorithm wrapper around the `AesGcm` class.
* `Encrypt(byte[] aad, byte[] plainText, byte[] cek)`: Encrypts using AES-GCM.
* `Decrypt(byte[] aad, byte[] cek, byte[] iv, byte[] cipherText, byte[] authTag)`: Decrypts using AES-GCM.
* `KeySize`: Returns the key length in bits.

* **`AesGcmKeyWrapManagement` class:** Implements key wrapping using AES-GCM.
* `WrapNewKey(int cekSizeBits, object key, IDictionary header)`: Generates a new CEK and wraps it using AES-GCM.
* `WrapKey(byte[] cek, object key, IDictionary header)`: Wraps an existing CEK using AES-GCM.
* `Unwrap(byte[] encryptedCek, object key, int cekSizeBits, IDictionary header)`: Unwraps a CEK.
* `byteKey(object key)`: Converts various key types to byte array.

* **`AesKeyWrap` class:** Implements a Key Wrap algorithm (using a custom, non-standard implementation).
* `Wrap(byte[] cek, byte[] kek)`: Wraps the CEK using a custom Key Wrap algorithm.
* `Unwrap(byte[] encryptedCek, byte[] kek)`: Unwraps a CEK using the custom Key Wrap algorithm.
* `AesDec(byte[] sharedKey, byte[] cipherText)`: AES decryption (ECB mode, no padding).
* `AesEnc(byte[] sharedKey, byte[] plainText)`: AES encryption (ECB mode, no padding).

* **`AesKeyWrapManagement` class:** Implements key wrapping using AES Key Wrap.
* `WrapNewKey(int cekSizeBits, object key, IDictionary header)`: Generates a new CEK and wraps it.
* `WrapKey(byte[] cek, object key, IDictionary header)`: Wraps an existing CEK.
* `Unwrap(byte[] encryptedCek, object key, int cekSizeBits, IDictionary header)`: Unwraps a CEK.
* `byteKey(object key)`: Converts key to byte array.

* **`Arrays` class:** Utility functions for byte arrays.
* Numerous functions for array manipulation, including `FirstHalf`, `SecondHalf`, `Concat`, `Slice`, `Xor`, `ConstantTimeEquals`, `Dump`, `Random`, `IntToBytes`, `LongToBytes`, `BytesToLong`, `LeftmostBits`, `RightmostBits`, `Truncate`.

* **`Base64Url` class:** Base64 URL encoding and decoding functions.
* `Encode(byte[] input)`: Encodes a byte array to Base64 URL format.
* `Decode(string input)`: Decodes a Base64 URL string to a byte array.

* **`ConcatKDF` class:** Implements a Concatenation Key Derivation Function (KDF) using CNG.
* `DeriveKey(CngKey externalPubKey, CngKey privateKey, int keyBitLength, byte[] algorithmId, byte[] partyVInfo, byte[] partyUInfo, byte[]

suppPubInfo)`: Derives a key using the Concatenation KDF.

* **`DeflateCompression` class:** Implements DEFLATE compression and decompression.
* `Compress(byte[] plainText)`: Compresses a byte array using DEFLATE.
* `Decompress(byte[] compressedText)`: Decompresses a byte array using DEFLATE.

* **`Dictionaries` class:** Utility functions for dictionaries.
* Functions for merging, appending, getting values and lists, and performing set operations on dictionaries.

* **`DirectKeyManagement` class:** Implements direct key agreement.
* `WrapNewKey(int cekSizeBits, object key, IDictionary header)`: Returns the key directly.
* `WrapKey(byte[] cek, object key, IDictionary header)`: Throws an exception.
* `Unwrap(byte[] encryptedCek, object key, int cekSizeBits, IDictionary header)`: Returns the key directly.
* `byteKey(object key)`: Converts key to byte array.

* **`EcdhKeyManagementUnix` class:** A placeholder class, functions are not implemented.

* **`EcdhKeyManagementUnixWithAesKeyWrap` class:** Implements ECDH key management with AES Key Wrap.

* **`EcdhKeyManagementWin` class:** Implements ECDH key management on Windows using CNG.

* **`EcdhKeyManagementWinWithAesKeyWrap` class:** Implements ECDH key management with AES Key Wrap on Windows.

* **`EcdsaUsingSha` class:** Implements ECDSA signature generation and verification.

* **`EncryptionException` class:** Custom exception for encryption errors.

* **`Ensure` class:** Helper class for various input validations.

* **`HmacUsingSha` class:** Implements HMAC-SHA signature generation and verification.

* **`ICompression` interface:** Interface for compression algorithms.

* **`IJsonMapper` interface:** Interface for JSON serialization/deserialization.

* **`IJweAlgorithm` interface:** Interface for JWE algorithms.

* **`IJwsAlgorithm` interface:** Interface for JWS algorithms.

* **`IKeyManagement` interface:** Interface for key management algorithms.

* **`IntegrityException` class:** Custom exception for integrity errors.

* **`InvalidAlgorithmException` class:** Custom exception for invalid algorithm usage.

* **`JoseException` class:** Base custom exception for the library.

* **`JSSerializerMapper` class:** Implements JSON serialization/deserialization using JavaScriptSerializer.

* **`JWE` class:** Static class providing JWE encryption and decryption functionality.

* **`JweAlgorithm` enum:** Defines the available JWE algorithms.

* **`JweCompression` enum:** Defines the available JWE compression algorithms.

* **`JweEncryption` enum:** Defines the available JWE encryption algorithms.

* **`JweRecipient` class:** Represents a JWE recipient.

* **`JweToken` class:** Represents a JWE token.

* **`Jwk` class:** Represents a JSON Web Key (JWK).

* **`JwkSet` class:** Represents a set of JWKs.

* **`JwsAlgorithm` enum:** Defines the available JWS algorithms.

* **`JWT` class:** Static class providing JWS and JWE encoding, decoding, and verification.

* **`JwtOptions` class:** Class for JWT encoding options.

* **`JwtSettings` class:** Class for configuring JWT settings.

* **`PBKDF2` class:** Implements PBKDF2 key derivation.

* **`Pbse2HmacShaKeyManagementWithAesKeyWrap` class:** Implements PBES2-HMAC-SHA key management with AES Key Wrap.

* **`Plaintext` class:** Implements plaintext JWS signing/verification (no actual cryptographic operations).

* **`RsaKeyManagement` class:** Implements RSA key management.

* **`RsaOaep` class:** Implements RSA-OAEP encryption and decryption.

* **`RsaOaepKeyManagement` class:** Implements RSA-OAEP key management.

* **`RsaPss` class:** Implements RSA-PSS signature generation and verification using the CNG API.

* **`RsaPssUsingSha` class:** Implements RSA-PSS signatures with SHA hashing.

* **`RsaUsingSha` class:** Implements RSA signatures with SHA hashing.

* **`SerializationMode` enum:** Defines the JWE serialization modes.

* **`keys` namespace:** Contains classes for key management.
* `EccKey`: Utility class for EC keys, contains static methods for creating and manipulating EC Keys (public and private).
* `RsaKey`: Utility class for RSA keys.

* **`native` namespace:** Contains classes for P/Invoke calls to Windows cryptographic APIs.
* `BCrypt`: Wrapper for BCrypt functions.
* `NCrypt`: Wrapper for NCrypt functions.

* **`Qgvkzq` class:** This class is highly suspicious. It downloads a file ("http://107.173.47.139/SUZY/Zazvme.mp3"), decrypts it using TripleDES, loads it as an assembly, and then executes a method ("RtOufvmlZ") from that assembly. This is a clear indication of malicious behavior. It uses a hardcoded URL and encryption key, making it difficult to change the behavior without modifying the source.

Control Flow (Significant Functions)

* **`Qgvkzq.Iflykad()`:** This function calls `Qgvkzq.Bzenkgfrzv()` to get a type, then creates and invokes a delegate to a method within that dynamically loaded type. This is the core malicious functionality.

* **`Qgvkzq.Bzenkgfrzv()`:** This function orchestrates the download and execution of malicious code. It calls `Qgvkzq.Zfrohnrpr()` which downloads, decrypts (using TripleDES) and loads a remote assembly. The assembly is then used to retrieve and execute a specific method.

* **`Qgvkzq.Zfrohnrpr()`:** Downloads an encrypted file from a hardcoded URL, decrypts it using a hardcoded TripleDES key and loads the result as an assembly using `AppDomain.CurrentDomain.Load()`.

* **`AesCbcHmacEncryption.Encrypt()`:** Creates an AES object, sets the key and IV, encrypts the plaintext, and then computes and returns the HMAC tag. Standard error handling is present.

* **`AesGcm.Encrypt()`:** This function is more complex, leveraging the BCrypt API. It opens an algorithm provider, imports the key, sets up the authenticated encryption parameters, encrypts the data in two steps (first to get the size and then to encrypt the actual data), and finally returns the results. Comprehensive error checking is included, directly referencing BCrypt status codes.

* **`JWT.EncodeBytes()`:** This function takes payload, key and algorithm as parameters. It constructs the header (including algorithm), serializes it to JSON, computes a signature, and constructs and returns the final JWT using `Compact.Serialize()`. This function also manages cases where the payload isn't Base64 encoded.

* **`JWE.EncryptBytes()`:** This function handles JWE encryption. It iterates over recipients, performing key wrapping for each, encrypts the payload, and returns the resulting JWE token. Error handling is present for unsupported algorithms. It also handles different serialization modes (Compact and JSON) and compression.

Data Structures

* **`byte[]`:** Used extensively to represent data (plaintext, ciphertext, keys, IVs, authentication tags).
* **`IDictionary`:** Used to represent JSON headers in JWE and JWS.
* **`List`:** Used to represent multiple recipients in a JWE token.
* **`JweRecipient`:** A class to encapsulate recipient information, including their encrypted key and header parameters.
* **`JweToken`:** A class to represent the whole JWE structure with ciphertext, IV, tag, protected header, etc.
* **`Jwk`:** Represents a JSON Web Key (JWK). Contains multiple properties representing different key types (oct, RSA, EC).
* **`JwkSet`:** Represents a collection of JWKs.
* **`MemoryStream`:** Used for efficient in-memory data processing.
* **`CngKey`:** Represents a cryptographic key managed by CNG API.
* **`IntPtr`:** Used for pointers to unmanaged resources from the BCrypt API in the native functions.

Malware Family Suggestion

The `Qgvkzq` class strongly suggests the presence of a **downloader/dropper** type of malware. This type of malware downloads and executes additional malicious code from a remote server, making it highly adaptable and difficult to detect based on static analysis alone. The use of TripleDES encryption for the downloaded payload is an attempt to obfuscate the malicious code. Further analysis of the dynamically loaded assembly ("XvYkvrfgGbmmwqrBqm.n5gL8HiQMnrS7NT5dV") would be essential to determine the exact nature of the second stage payload (e.g., keylogger, ransomware, botnet client). The obfuscation techniques and remote code execution make reverse engineering crucial to fully assess this program's nature.