CryptionSlater

Generic Symmetric Encryption and HMAC Library v1

.NET/C# 4+

Driver's Manual

Logan Gore

About the Developer

Logan Gore is a software engineer who has published several Open Source projects for C#/.NET and papers on current industry topics such as enterprise architecture, cryptography, and Big Data. He holds professional certifications in enterprise security, software and database development, cloud technologies, networking, and project management. Logan is currently involved in cloud, game, and mobile development.

loganlgore@gmail.com

Part 1 – Intro to CryptionSlater

What is CryptionSlater?

CryptionSlater is a generic encryption and HMAC library that gives developers a simple and secure way to implement symmetric encryption and message authentication in .NET/C# 4+. CryptionSlater will help you avoid the errors commonly found in cryptographic solutions by providing a clean and intuitive interface. Even if you are new to C# or encryption, CryptionSlater will get you started quickly and with confidence.

If You Are New To Encryption

CryptionSlater is easily implemented by the novice programmer, but designing a secure solution will involve much more than the examples and topics covered in this manual. It's highly recommended that newcomers research both the theoretical and practical aspects of cryptography and secure development in your language and framework; in our case, this is .NET/C#.

CryptionSlater Features

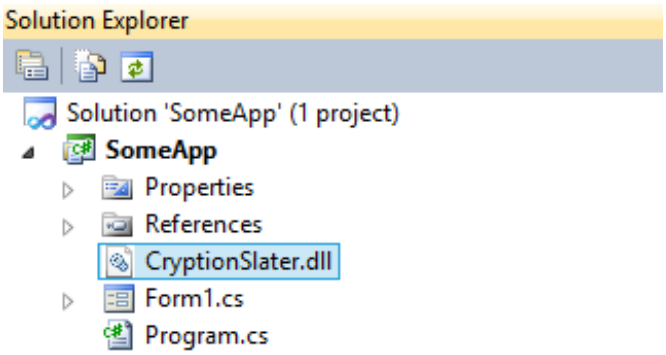
Out-the-box, CryptionSlater offers generic implementation of the System.Security.Cryptography **SymmetricAlgorithm** and **HMAC** subclasses, random salts, chained PBKDF2 stretching, and will default to the strongest key and block sizes offered by the specified algorithm. Cipher and padding modes are set to CBC and PKCS7 (respectively) and cannot be adjusted. CryptionSlater also has custom features to make the developer's life easier, like auto adjusting stretch iterations until a targeted timeframe has been met.

You shouldn't use CryptionSlater if…

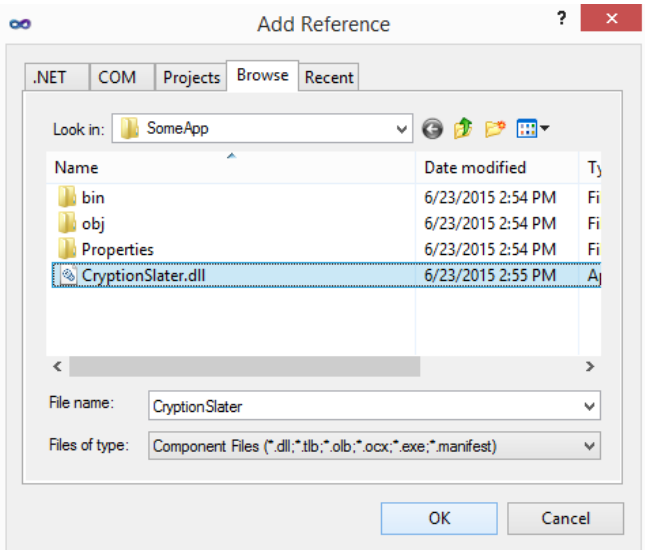
CryptionSlater is great because it's easy to implement without shooting yourself in the foot. But the developer will have less control than when writing their own encryption scheme or wrapper class. For instance, HMAC is performed by default and cannot be disabled, and salts and stretch iterations are attached to the ciphertext. These are all things that should be considered before implementing CryptionSlater in a production environment.

Adding the CryptionSlater DLL to Your Project

First, add the DLL to the project:



Next, add CryptionSlater to the project references:



Finally, use the **using** directive to reference CryptionSlater as well as **System.Security.Cryptography** in the .cs file:

```
using CryptionSlater;
using System.Security.Cryptography;
```

Part 2 –The CryptionSlater Class

2.1 Getting Started

String Encryption Example

Basic steps for using CryptionSlater are as follows:

1. Create an instance of the CryptionSlater class. Specify the algorithms in the generic type parameters and the keys in the constructor (use your own strong keys, not ours from the example).
2. Use the encryption or decryption methods.

Here we will use **RijndaelManaged** for our encryption algorithm and **HMACSHA256** as our HMAC. The encryption algorithm is used in the first generic type parameter and the HMAC algorithm in the second. Both keys must be provided in the constructor and can be either byte[] or string format. The first constructor parameter is for the encryption key, the second is for the authentication key.

```
CryptionSlater<RijndaelManaged, HMACSHA256> cs = new CryptionSlater<RijndaelManaged,
HMACSHA256>("f57(%Gkjh5g897&", "EnNN84@$#fV>JWGnr");

string plaintext = "hello, this is a secret.";

//encrypt
string encryptedData = cs.EncryptToBase64(plaintext);

//decrypt
string decryptedData = cs.DecryptBase64ToString(encryptedData);
```

Choosing Your Encryption Algorithm

CryptionSlater can only be used with symmetric block ciphers that inherit from the **SymmetricAlgorithm** base class in .NET. The default .NET algorithms can be found in the **System.Security.Cryptography** namespace.

White-listed (secure) .NET algorithms included in the **System.Security.Cryptography** namespace:

Name (class)	Block Size (bits)	Key Size (bits)
AesManaged	128	128 ,192, 256
AesCryptoServiceProvider	128	128, 192, 256
RijndaelManaged	128, 192, 256	128, 192, 256

Choosing Your HMAC Algorithm

CryptionSlater can only be used with HMACs that inherit from the HMAC base class in .NET. The default .NET algorithms can be found in the **System.Security.Cryptography** namespace. For best security **HMACSHA256** or greater is recommended.

Keys

You need two keys (passwords) to use CryptionSlater; one for encryption and one for authentication. Keys can be byte[] or string format. Refer to the *Key Management* section in this guide for more detailed information about key management.

The **GetBytes** method of the **RNGCryptoServiceProvider** class in the **System.Security.Cryptography** namespace can be used to generate cryptographically secure random byte[] material to use as keys (this only handles generation, the developer will still have to take care of key management).

> **Note:** Make sure to read the .NET Strings section in Part 4 if you are considering using string keys.
>
> **Note:** As long as salting and stretching is being used, the keys provided in the constructor do not need to be the exact same size as the key bit size of the algorithm (and will never actually touch the algorithm). Otherwise the stretching function will derive keys to the correct size. This means that if your algorithm is using 256 bit keys, you could specify a key in the constructor that isn't 256 bits but the key derivation (stretching) function will derive a key that is correctly sized at 256 bits.

Constructors

CryptionSlater(byte[],byte[])	Creates an instance of CryptionSlater using two byte[] keys.
CryptionSlater(byte[],byte[],CryptionSlaterMode)	Creates an instance of CryptionSlater using two byte[] keys and a CryptionSlaterMode.
CryptionSlater(string,string)	Creates an instance of CryptionSlater using two string keys.
CryptionSlater(string,string,CryptionSlaterMode)	Creates an instance of CryptionSlater using two string keys and a CryptionSlaterMode.

Public Methods

Method Name	**Description**
DecryptBase64ToBytes(string)	Decrypts and authenticates a ciphertext string in Base64 format to a byte[].
DecryptBase64ToString(string)	Decrypts and authenticates a ciphertext string in Base64 format to a UTF8 encoded string.
DecryptToBytes(byte[])	Decrypts and authenticates a ciphertext in byte[] format to a byte[].
EncryptToBase64(byte[])	Encrypts and HMACs a byte[] to a Base64 formatted string.
EncryptToBase64(string)	Encrypts and HMACs a string to a Base64 formatted string.
EncryptToBytes(byte[])	Encrypts and HMACs a byte[] to a byte[].

Properties

Property Name	Type	Description
AuthenticationKeyStretchTargetMS	int	The approximate time (milliseconds) the stretching function for the authentication key derivation if CryptionSlaterMode.AutoAdjust is being used. Default is 200ms.
AutoAdjustEncryptionKeyStretching	bool	Manually enables or disables the auto adjust mode for a single key (see *AutoAdjust* under *CryptionSlaterMode* in section 2.2).
AutoAdjustAuthenticationKeyStretching	bool	Manually enables or disables the auto adjust mode for a single key (see *AutoAdjust* under *CryptionSlaterMode* in section 2.2).
BlockBitSize	int	The cipher's block size in bits. Defaults to the largest allowable for the encryption algorithm.
EncryptionKeyStretchTargetMS	int	The approximate time (milliseconds) the stretching function will run for encryption key derivation if CryptionSlaterMode.AutoAdjust is being used. Default is 200ms.
KeyBitSize	int	The cipher's key size in bits. Defaults to the largest allowable for the encryption algorithm (this size will also be used to derive the authentication keys unless salting and stretching is turned off).
LastEncryptionKeyStretchingTime	TimeSpan	The time it took to stretch the encryption key during the last encryption phase.
LastAuthenticationKeyStretchingTime	TimeSpan	The time it took to stretch the authentication key during the last encryption phase.
SaltBitSize	Int16	The salt size in bits. Defaults to the KeyBitSize. Must be at least 64 bits (8 bytes).
StretchIterations	Int32	The number of iterations the chained PBKDF2 stretching function will run. Default is 6000. Setting this property to anything less than 1 will result in no salts being used and no stretching being performed.

		If CryptionSlaterMode.AutoAdjust is being used, this number will have no effect on the stretching function and will be set to reflect the iterations that the AutoAdjust used the last time the encryption function was called.

Text Encoding

WARNING: The string type in .NET is not considered secure for cryptographic purposes, make sure to read .NET Strings in Part 4 if you are considering handling string data (keys or plaintext).

The methods in CryptionSlater that handle string data use UTF8 encoding (System.Text) to convert to and from byte[] format before encryption and after decryption – not in the middle. Base64 is used in between encryption and decryption for transmitting or storing encrypted data as text when using EncryptToBase64, DecryptBase64ToBytes, EncryptToBase64, and DecryptBase64ToString.

Be careful if introducing additional text encoding. Do not use the .NET encoding class to convert *encrypted* data; if using the encoding class to perform string or byte conversions, do it before encryption and after decryption –not in the middle. Use the EncryptToB64 method if there is a need for the encrypted data to be transmitted as text, or stored in a database as text (encrypted to a string format).

Keep in mind, Base64 formatting will consume 1/3 more storage.

2.2 Settings and Tuning

CryptionSlaterMode

CryptionSlater has an enum called **CryptionSlaterMode**. This makes it easier to change the performance and security of your cryptographic solution. The mode is used as input to determine the number of chained PBKDF2 stretch iterations that will be used to derive the encryption and authentication keys. **CryptionSlaterMode** can only be used in the CryptionSlater class constructor. If not specified the mode will default to *Balanced*.

Mode	**PBKDF2 Iterations**
HighPerformance	10
Performance	1000
Balanced	6000
Secure	30000
AutoAdjust	Adjusts iterations based on stretching time in MS that is set in **AutoAdjustAuthenticationKeyStretching** and **AutoAdjustEncryptionKeyStretching** properties. Default is 200ms. (This assumes DateTime.Now is working correctly.)

Note: HighPerformance mode will compute the fastest but offer less security. Conversely, **Secure** mode will be more secure but impractical for most applications because of how long it will take to compute. The developer can always specify the stretch iterations through the **EncryptionKeyStretchIterations** and **AuthenticationKeyStretchIterations** properties if they want more control than what the **CryptionSlaterMode** provides. See the *StretchIterations* section for more information on adjusting stretching iterations and best practices.

AutoAdjust

The **AutoAdjust** mode will stretch for a designated time rather than a designated number of iterations. This will help ensure that changes made to the settings within CryptionSlater or changes made to physical hardware, like CPUs, will not reduce the stretching time. The default is set to 200ms. This means that if **AutoAdjust** mode is used, the keys will stretch until the threshold time is reached. This does, however, rely on DateTime.Now and assumes it will be working correctly. Keep in mind, the number of iterations used to derive the encryption key could take more or less time when computing for the authentication key; 10% is normal.

The **LastAuthenticationKeyStretchingTIme** and **LastEncryptionKeyStretchingTime** properties can be used to get the TimeSpan of the last actual key stretching times.

Key Size

Key sizes are adjustable in CryptionSlater using the **KeyBitSize** propery and must be allowable by the specified algorithm. For best security, you should use the largest keys possible. By default, CryptionSlater will use the largest key size allowed by the algorithm. The largest key in **RijndaelManaged** is 256 bits. CryptionSlater would therefore use this by default. But a developer could easily adjust the key size to 128 bits if they wanted:

```
CryptionSlater<RijndaelManaged, HMACSHA256> cs = new …

cs.KeyBitSize = 128;
```

It should be noted that the block size will still default to the largest allowable. To find valid key sizes for any algorithm inheriting from **SymmetricAlgorithm**, use the **LegalKeySizes** property.

Note: Aes and Rijndael will both support 128, 192, and 256 bit keys.

Note: The same key bit size will be used for the encryption and authentication unless no salting or stretching is being performed.

As long as salting and stretching is being used, the keys provided in the constructor do not need to be the exact same size as the key bit size of the algorithm. Otherwise the stretching function will correctly size the derived keys. This means that if your algorithm is using 256 bit keys, you could specify a key in the constructor that isn't 256 bits but the key derivation (stretching) function will derive a key that is correctly sized at 256 bits. This is another benefit of using salting and stretching in CryptionSlater.

Block Size

Block sizes follow the same rules as key sizes and can set by the developer with the **BlockBitSize** property. To find valid block sizes for any algorithm inheriting from **SymmetricAlgorithm**, use the **LegalBlockSizes** property.

Like in the Key Size example, the BlockBitSize can be set manually:

```
CryptionSlater<RijndaelManaged, HMACSHA256> cs = new …

cs.BlockBitSize = 128;
```

> **Note:** Aes classes only support 128 bit blocks and will always default to 128. Rijndael will support 128, 192, or 256, and will default to 256.

SaltSize

Salt sizes will be defaulted to the key size and can be adjusted by the developer, but it is not recommended to reduce the salt size below that of the key size. If manually setting both the key and salt sizes, the salt size should be set after the key size, otherwise the key size will overwrite the salt size. The SaltSize property must be set to at least 64 bits (8 bytes).

Stretch Iterations

The **EncryptionKeyStretchIterations** and **AuthenticationKeyStretchIterations** properties (Int32) control the number of stretching iterations the chained PBKDF2 function runs when deriving the encryption and authentication keys. These properties can be set directly, or by using the **CryptionSlaterMode** enum in the class constructor.

As a best practice, stretching iterations should take between 200 and 1000ms to compute. You might ask, well how am I supposed to figure that out? The **CryptionSlaterMode.AutoAdjust** will figure it out for you. Using this mode will default to 200ms, or the developer can specify the time **AutoAdjust** mode should run (see the *AutoAdjust* section under *CryptionSlaterMode*). The convenient **LastAuthenticationKeyStretchingTIme** and **LastEncryptionKeyStretchingTime** properties can also be used to get the **TimeSpan** of the last actual key stretching times to help with tuning or debugging

Below we could set AutoAdjust to true for the encryption key and specify exactly 10000 iterations for the authentication key. This means the encryption key would stretch for approximately 200ms while the authentication key would stretch for 10000 iterations regardless of how long it took.

```
CryptionSlater<RijndaelManaged, HMACSHA256> cs = new …

cs.AutoAdjustEncryptionKeyStretching = true;
cs.AuthenticationKeyStretchIterations = 10000;
```

But we might want to have the encryption key stretching take 300ms:

```
cs.EncryptionStretchKeyTargetMS = 300;
```

Note: Setting either of the stretching iterations properties to 0 (actually anything less than 1) will disable salting and stretching for that particular key.

Portability Considerations

In addition to the encrypted data and the HMAC, CryptionSlater ciphertexts will include the salts, stretching iterations, and the salt size. This means that developers will be able to adjust their stretching iterations at will without worrying about future compatibility issues. Developers, however, must make sure that they use the same algorithms and the same key sizes, and block sizes (this is all the more reason to take advantage of the sane defaults). Key strength criteria should also be the same if it has been made more restrictive (see Part 3).

2.3 Recommended Configurations

Property/Parameter	**Setting**
Encryption Algorithm*	Aes or Rijndael
HMAC Algorithm	HMAC-SHA series. 256 or larger is preferable.
Keys	Highly random and at least the full bit length of –or greater than– the KeyBitSize property.
KeyBitSize	Default
BlockBitSize	Default
StretchIterations/CryptionSlaterMode	Balanced (Default) or AutoAdjust. This setting will have the largest impact on performance and will have to be adjusted at the developer's discretion given the environment and security requirements.
EncryptionKeyStretchTargetMS / AuthenticationKeyStretchTargetMS	Default
SaltBitSize	Default

*If you are trying to implement AES256, you need to use an AES implementation, 256-bit keys, and a 128-bit block.

*Cipher and padding modes are defaulted to CBC and PKCS7 (respectively).

Part 3 – The KeyStrength Class

The CryptionSlater class itself doesn't perform any key checking. Enter the key strength class. This class is used to check the strength of a key or keys using regular expressions.

Constructors

KeyStrength()	Defaults the regular expression used to validate key strength to `@"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{8,}"`
KeyStrength(string)	Sets the regular expression used to validate key strength.

Public Methods

Name	**Description**
IsStrongKey(byte[] + (string)	Returns a bool indicating if the supplied key passed the regular expression (regex set optionally in the constructor).
AreKeysSecure(byte[],byte[]) + (string,string)	Returns a bool indicating if the supplied keys 1. Are not null; 2. Do not match; and 3. Pass regular expression checking.

Part 4 –String Immutability, Key Management, and Tips to Remember

.NET Strings

Developers working with string data in a .NET/C# environment should understand string immutability issues. If a string in .NET is changed, the old string will still exist in memory until garbage collection, making it impossible to perform secure memory management of this data. .NET's nondeterministic memory management means that sensitive data could even reside in memory for longer than expected, get relocated to different parts of memory, or even be subject to paging. byte[] types are strongly recommended for handling secure data.

Key Management

Key management is the most challenging – and risky- aspect of implementing a symmetric encryption system. Unfortunately, developers will oftentimes worry about algorithm strength and spend little time on key security. It is worth doing the research to look for secure key and memory management solutions available to your framework.

The following key management issues should always be addressed *prior* to implementing your system:

- Generation
  - How will the keys be created?
  - Will this method generate strong keys?
- Distribution
  - How will keys be transmitted securely to parties that need them (if this is necessary)?
- Installation/Usage
  - In what environments will the keys actually be installed and used?

    - Assess the security of those environments.
- Storage
    - What steps will need to be taken to safely store keys?
    - Is the level of security that protects the keys greater than or equal to the security of the encryption? If not, it could be easier for an attacker to compromise your key storage (stealing the keys) than to guess/crack your keys.
    - Some people hard code their keys on a server or on an app that resides on their server. Others opt for software that manages and secures keys (usually a better option). These decisions are often platform and even machine specific. It is always a good idea to look for secure memory solutions that the OS or platform can provide.
- Control/Recovery
    - Who will have control/access to the keys?
    - What happens if the person in control of the keys leaves?
- Change
    - For best security, keys should be changed. How often, is specific to the security model.
    - If keys are changed, how will old keys be stored, managed, and/or (if needed) safely reintroduced to the encryption and decryption process?
- Disposal
    - Have a secure process for disposing of keys and the memory they have resided on.

Tips to Remember

1. USE STRONG KEYS.
2. Do not lose the keys, the encrypted data will likely be unrecoverable.
3. Be careful with text encoding. Do not use the .NET encoding class to convert *encrypted* data; if using the encoding class to perform string or byte conversions, do it before encryption and after decryption –not in the middle.
4. Do not change your keys in an application where you need to decrypt data that was previously encrypted using an old/different key (unless you have a key management system).
5. Key storage can be a serious vulnerability in a symmetric key system. Because key storage is outside the scope of the CryptionSlater dll, obtaining keys is one of the most effective attacks to compromise the security it provides. Developers should consider evaluating the security/storage of their keys and the memory those keys reside on. Scan the server/system for vulnerabilities and use strong passwords.
6. Handle application errors and make sure no secret information (or any information that could help compromise your data) is revealed in error messages.