Crypto Class

Provides methods for creating digests, message authentication codes, and signatures, as well as encrypting and decrypting information.

Namespace

System

Usage

The methods in the `Crypto` class can be used for securing content in Lightning Platform, or for integrating with external services such as Google or Amazon WebServices (AWS).

Each method in this class supports a unique set of AES encryption algorithms, depending on its purpose. To confirm which algorithms are available for the action that you want to do, check each method.

Using Encryption Algorithms

The Crypto class supports Galois Counter Mode (GCM) and Cipher Block Chaining mode (CBC). The GCM AES256-GCM algorithm is valid in all encrypt and decrypt variants Currently, only the 256-bit size is supported for GCM. CBC algorithms in 128-bit, 192-bit, and 256-bit sizes are valid in all variants except those that expect additional authentication data (an *aaD* parameter).

- When you use CBC with encrypt and decrypt, you provide a 16-bit initialization vector (IV).
- When you use GCM with encrypt and decrypt, you provide no initialization vector (IV).
- When you use CBC with encryptWithManagedIV and decryptWithManagedIV, Salesforce provides the IV. You provide no additional authentication data (aaData). You can only use CBC with encryptWithManagedIV and decryptWithManagedIV if you use the version which does not expect aaData.
- When you use GCM with encryptWithManagedIV and decryptWithManagedIV, Salesforce provides an IV. and you can optionally provide the aaData

When use the Crypto class to encrypt using GCM, the final encrypted content includes the length of the IV (always 12), the Salesforce-generated 12-byte IV, and the cipher text.

Encryption Algorithms

The Crypto class supports these encryption algorithms.

MODE	VARIANT	DESCRIPTION
CBC (Cipher Block Chaining)	`AES128`, `AES128-CBC`	AES 128-bit with CBC mode with PKCS7 padding. Use either of these two values.
	`AES192`, `AES192-CBC`	AES 192-bit with CBC mode with PKCS7 padding. Use either of these two values.

GCM (Galois Counter Mode)	`AES256-GCM`	AES 256-bit with GCM mode with no padding. Currently, only the 256-bit size is supported for GCM.

Signing Algorithms

The Crypto class supports these signing algorithms.

TYPE	VARIANT	DESCRIPTION
RSA	`RSA`, `RSA-SHA1`	An RSA signature (with an asymmetric key pair) of an SHA1 hash. Use either of these two values.
	`RSA-SHA256`	RSA signature of an SHA256 hash
	`RSA-SHA384`	RSA signature of an SHA384 hash
	`RSA-SHA512`	RSA signature of an SHA512 hash
ECDSA (DER)	`ECDSA-SHA256`	ECDSA signature of an SHA256 hash
	`ECDSA-SHA384`	ECDSA signature of an SHA384 hash
	`ECDSA-SHA512`	ECDSA signature of an SHA512 hash
ECDSA (P1363)	`ECDSA-SHA256-P1363`	ECDSA signature of an SHA256 hash (P1363 format)
	`ECDSA-SHA256-PLAIN`	ECDSA signature of an SHA256 hash (P1363 format). Use if the JWT returns invalid_client. See Other Errors on this page.
	`ECDSA-SHA384-P1363`	ECDSA signature of an SHA256 hash (P1363 format)
	ECDSA-SHA512-P1363	

Encrypt and Decrypt Exceptions

These exceptions can be thrown for these methods.

- `decrypt`
- `encrypt`
- `decryptWithManagedIV`
- `encryptWithManagedIV`

Exception	Message	Description
`InvalidParameterValue`	Unable to parse the initialization vector from encrypted data.	Thrown if you're using managed initialization vectors, and the cipher text is less than 16 bytes.

	the suported AES algorithms listed on this page.	
	Invalid private key. Must be *size* bytes.	Thrown if the size of the private key doesn't match the specified algorithm.
	Invalid initialization vector. For CBC, this must be 16 bytes. For GCM, the IV is 12 bytes.	Thrown if the initialization vector provided for a CBC encryption isn't 16 bytes.
	AAD can only be used with AESGCM algorithms.	Thrown if a value is supplied for *aaData*, but the encryption algorithm isn't a GCM type.
	Invalid data. Input data is *size* bytes, which exceeds the limit of 1,048,576 bytes.	Thrown if the data is greater than 1 MB. For decryption, 1,048,608 bytes are allowed for the initialization vector header, plus any additional padding the encryption added to align to block size.
`NullPointerException`	*Argument* can't be null.	Thrown if one of the required method arguments is null.
`SecurityException`	Given final block isn't properly padded.	Thrown if the data isn't properly block-aligned or similar issues occur during encryption or decryption.
`SecurityException`	*Message Varies*	Thrown if something goes wrong during either encryption or decryption.

These exceptions are a subset of the exceptions that can be thrown from the System namespace. Refer to Exception Class and Built-In Exceptions

For CBC, the `Crypto` class uses AES / CBC / PKCS7 padding, which is vulnerable to a Padding Oracle attack. You can protect against a Padding Oracle attack by using the Encrypt-then-MAC method. In this method, you encrypt the cipher text and MAC separately.

- For encryption, first encrypt the data with AES by using one encryption key. Then, with a different encryption key, use the generateMac(algorithmName, input, privateKey) method to generate a message authentication code (MAC) for the cipher text. Append the MAC to the cipher text before sending it to its recipient.
- For decryption, start by checking the authenticity and integrity of the cipher text by using the verifyHMac(algorithmName, data, privateKey, macToVerify) method. If either the authenticity or integrity check fails, throw an exception and don't decrypt the cipher text. The decryption of the cipher text must only happen in a second step, after the message authenticity and integrity has been verified.

You can also protect against a Padding Oracle attack by using a GCM encryption algorithm.

Other Errors

Under rare conditions you may encounter the `invalid_client` error from the JSON Web Tokens (JWT) service.

Error	Message	Description
`invalid_client`	The actual text varies, but describes the inability to validate the client credentials.	The JWT public certificate in the Salesforce Connected Application doesn't appear to match the known private key.

For example, in order to comply with your program requirements, you sign your token using the Elliptic Curve Digital Signature Algorithm (ECDSA) with the P-256 curve. This algorithm is in the P1363 format, so when you try to use `Crypto.verify()` using the `ECDSA SHA256`, you receive a response containing `invalid_client`. You change `ECDSA-SHA256` to `ECDSA-SHA256-PLAIN` and the error is resolved.

See Also

- Encrypt-then-MAC (EtM)
- ISO/IEC 19772:2020 - Information Security Authenticated Encryption
- Exception Class and Built-In Exceptions

Crypto Methods

The following are methods for `Crypto`. All methods are static.

Crypto Methods

- **decrypt(algorithmName, secretKey, initializationVector, cipherText)**
  Decrypts the *cipherText* blob by using the specified algorithm, private key, and initialization vector. Use this method to decrypt blobs encrypted by using a third-party application or the `encrypt` method.
- **decryptWithManagedIV(algorithmName, secretKey, IVAndCipherText)**
  Decrypts the *IVAndCipherText* blob by using the specified algorithm and private key. Use this method to decrypt blobs encrypted by using a third-party application or the `encryptWithManagedIV` method. This version of decryptWithManagedIV doesn't use additional authentication data.
- **decryptWithManagedIV(algorithmName, secretKey, IVAndCipherText, aaData)**
  Decrypts the *IVAndCipherText* blob by using the specified algorithm and private key. Use this method to decrypt blobs encrypted by using a third-party application or the `encryptWithManagedIV` method. This version of decryptWithManagedIV uses additional authentication data. CBC isn't supported for this method.
- **encrypt(algorithmName, secretKey, initializationVector, clearText)**
  Encrypts the *clearText* blob by using the specified algorithm, private key, and initialization vector. Use this method when you want to specify your own initialization vector.
- **encryptWithManagedIV(algorithmName, secretKey, clearText)**
  Encrypts the *clearText* blob by using the specified algorithm and private key. Use this method when you want Salesforce to generate the initialization vector. This version of encryptWithManagedIV doesn't use additional authentication data.
- **encryptWithManagedIV(algorithmName, secretKey, clearText, aaData)**
  Encrypts the *clearText* blob by using the specified algorithm and private key. Use this method when you want Salesforce to generate the initialization vector. This version of encryptWithManagedIV uses additional authentication data. CBC isn't supported for this method.
- **generateAesKey(size)**
  Generates an Advanced Encryption Standard (AES) key.
- **generateDigest(algorithmName, input)**
  Computes a secure, one-way hash digest using the specified algorithm on the supplied *input* blob.
- **generateMac(algorithmName, input, privateKey)**
  Computes a message authentication code (MAC) for the *input* blob value using the private key and the specified algorithm.
- **getRandomInteger()**
  Returns a random integer value.
- **getRandomLong()**
  Returns a random long value.

Computes a unique digital signature for the input blob value, using the specified algorithm and the supplied certificate and key pair.

- **signXML(algorithmName, node, idAttributeName, certDevName)**
  Envelops the signature into an XML document.
- **signXML(algorithmName, node, idAttributeName, certDevName, refChild)**
  Inserts the signature envelope before the specified child node.
- **verify(algorithmName, data, signature, publicKey)**
  Verifies the digital signature for the *data* blob using the specified algorithm and the supplied public key. Use this method to verify a blob signed by a digital signature created using a third-party application or the sign method.
- **verify(algorithmName, data, signature, certDevName)**
  Verifies the digital signature for the *data* blob using the specified algorithm and the public key associated with *certDevName*. Use this method to verify a blob signed by a digital signature created using a third-party application or the `signWithCertificate` method.
- **verifyHMac(algorithmName, data, privateKey, macToVerify)**
  Verifies the HMAC signature for the *data* blob using the specified algorithm, input data, private key, and the mac. Use this method to verify a blob signed by a digital signature created using a third-party application or the sign method.

decrypt(algorithmName, secretKey, initializationVector, cipherText)

Decrypts the *cipherText* blob by using the specified algorithm, private key, and initialization vector. Use this method to decrypt blobs encrypted by using a third-party application or the `encrypt` method.

Signature

```
public static Blob decrypt(String algorithmName, Blob secretKey, Blob initializationVector, Blob cipherText)
```

Parameters

algorithmName

Type: String

decrypt supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.

- `AES128, AES128-CBC`
- `AES192, AES192-CBC`
- `AES256, AES256-CBC`
- `AES256-GCM`

secretKey

Type: Blob

Private key text. The length of *secretKey* must match the size required by *algorithmName*: 128 bits, 192 bits, or 256 bits, which is 16 bytes, 24 bytes, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key.

initializationVector

Type: Blob

- For CBC, the 128 bit (16 byte) IV. The IV must be 128 bits (16 bytes.)
- For GCM, don't provide an IV. Any non-null IV will result in an error.

cipherText

Type: Blob

Contains the decrypted contents of *cipherText*.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestDecrypt {

        public void testDecrypt(){
            // 16-byte string
            Blob exampleIv = Blob.valueOf('Example of IV123');
            Blob key = Crypto.generateAesKey(128);
            Blob data = Blob.valueOf('Data to be encrypted');
            Blob encrypted = Crypto.encrypt('AES128', key, exampleIv, data);

            Blob decrypted = Crypto.decrypt('AES128', key, exampleIv, encrypted);
            String decryptedString = decrypted.toString();
            System.debug('Decrypted Value: ' + decryptedString);
            Assert.areEqual('Data to be encrypted', decryptedString, 'Error: not equal!');
            return;
        }
    }
```

To invoke this method, run:

```
TestDecrypt td = new TestDecrypt();
td.testDecrypt();
```

decryptWithManagedIV(algorithmName, secretKey, IVAndCipherText)

Decrypts the *IVAndCipherText* blob by using the specified algorithm and private key. Use this method to decrypt blobs encrypted by using a third-party application or the `encryptWithManagedIV` method. This version of decryptWithManagedIV doesn't use additional authentication data.

Signature

```
public static Blob decryptWithManagedIV(String algorithmName, Blob secretKey, Blob IVAndCipherText)
```

Parameters

algorithmName

Type: String

decryptWithManagedIV supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.

- `AES128`, `AES128-CBC`
- `AES192`, `AES192-CBC`
- `AES256`, `AES256-CBC`
- `AES256-GCM`

secretKey

Type: Blob

Type: Blob

A concatenation of the initialization vector and the encrypted text that you want to decrypt.

- For CBC, *IVAndCipherText* must contain IV + ciphertext, where the IV must be the first 128 bits (16 bytes) with the ciphertext following.
- FOR GCM, *IVAndCipherText* must contain the length of the IV (always 12) followed by a 96 bit (12 byte) IV, with the ciphertext following.

Return Value

Type: Blob

Contains the decrypted contents of *IVAndCipherText*.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestDecryptWithManagedIV {

        public void testDecryptWithManagedIV(){
            String algorithmName = 'AES128';
            Blob key = Crypto.generateAesKey(128);
            Blob data = Blob.valueOf('Data to be encrypted');
            Blob encrypted = Crypto.encryptWithManagedIV(algorithmName, key, data);
            Blob decrypted = Crypto.decryptWithManagedIV(algorithmName, key, encrypted);
            String decryptedString = decrypted.toString();
            Assert.areEqual('Data to be encrypted', decryptedString, 'Error: the strings a

        }
    }
```

To invoke this method, run:

```
TestDecryptWithManagedIV tdiv = new TestDecryptWithManagedIV();
tdiv.testDecryptWithManagedIV();
```

decryptWithManagedIV(algorithmName, secretKey, IVAndCipherText, aaData)

Decrypts the *IVAndCipherText* blob by using the specified algorithm and private key. Use this method to decrypt blobs encrypted by using a third-party application or the `encryptWithManagedIV` method. This version of decryptWithManagedIV uses additional authentication data. CBC isn't supported for this method.

Signature

```
public static Blob decryptWithManagedIV(String algorithmName, Blob secretKey, Blob
IVAndCipherText, Blob aaData)
```

Parameters

algorithmName

Type: String

decryptWithManagedIV supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.

Private key text. The length of *secretKey* must match the size required by *algorithmName*: 128 bits, 192 bits, or 256 bits, which is 16 bytes, 24 bytes, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key for you.

IVAndCipherText

Type: Blob

IVAndCipherText must contain the length of the IV (always 12) followed by a 96 bit (12 byte) IV, with the ciphertext following.

aaData

Type: Blob

Additional authentication data. This value is required.

Return Value

Type: Blob

Contains the decrypted contents of *IVAndCipherText*.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestDecryptWithManagedIV {

    public void testDecryptWithManagedIV(){
        String algorithmName = 'AES256-GCM';
        Blob key = Crypto.generateAesKey(256);
        Blob data = Blob.valueOf('Data to be encrypted');
        Blob aad = Blob.valueOf('Additional tag');
        Blob encrypted = Crypto.encryptWithManagedIV(algorithmName, key, data, aad);
        Blob decrypted = Crypto.decryptWithManagedIV(algorithmName, key, encrypted, aa
        String decryptedString = decrypted.toString();
        Assert.areEqual('Data to be encrypted', decryptedString, 'Error: the strings a

    }
}
```

To invoke this method, run:

```
TestDecryptWithManagedIV tdiv = new TestDecryptWithManagedIV();
tdiv.testDecryptWithManagedIV();
```

encrypt(algorithmName, secretKey, initializationVector, clearText)

Encrypts the *clearText* blob by using the specified algorithm, private key, and initialization vector. Use this method when you want to specify your own initialization vector.

Signature

```
public static Blob encrypt(String algorithmName, Blob secretKey, Blob initializationVector,
Blob clearText)
```

Parameters

algorithmName

Type: String

- `AES192, AES192-CBC`
- `AES256, AES256-CBC`
- `AES256-GCM`

secretKey

Type: Blob

Private key text. The length of *secretKey* must match the size required by *algorithmName*: 128 bits, 192 bits, or 256 bits, which is 16 bytes, 24 bytes, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key for you.

initializationVector

Type: Blob

- For CBC, any 128 bit (16 byte) string to provide the initial state to this method. The initialization vector must be 128 bits (16 bytes.)
- For GCM, don't provide an IV. Any non-null IV results in an error.

clearText

Type: Blob

The content that you want to encrypt.

Return Value

Type: Blob

Contains the encrypted contents of *clearText*.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestEncrypt {

        public void testEncrypt(){
            Blob exampleIv = Blob.valueOf('Example of IV123');
            Blob key = Crypto.generateAesKey(128);
            Blob data = Blob.valueOf('Encryption Example Text.');
            Blob encrypted = Crypto.encrypt('AES128', key, exampleIv, data);

            Blob decrypted = Crypto.decrypt('AES128', key, exampleIv, encrypted);
            String decryptedString = decrypted.toString();
            Assert.areEqual('Encryption Example Text.', decryptedString, 'Error: the value
            return;
        }
    }
```

To invoke this method, run:

```
TestEncrypt te = new TestEncrypt();
te.testEncrypt();
```

encryptWithManagedIV(algorithmName, secretKey, clearText)

Encrypts the *clearText* blob by using the specified algorithm and private key. Use this method when you want Salesforce to generate the initialization vector. This version of encryptWithManagedIV

Parameters

algorithmName

Type: String

encryptWithManagedIV supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.

- `AES128`, `AES128-CBC`
- `AES192`, `AES192-CBC`
- `AES256`, `AES256-CBC`
- `AES256-GCM`

secretKey

Type: Blob

Private key text. The length of *secretKey* must match the size required by *algorithmName*: 128 bits, 192 bits, or 256 bits, which is 16 bytes, 24 bytes, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key for you.

clearText

Type: Blob

The content you want to encrypt.

Return Value

Type: Blob

Contains the encrypted contents of *clearText*.

- For CBC, the initialization vector is stored as the first 128 bits (16 bytes) of the encrypted blob.
- For GCM, the return value contains the length of the IV (always 12) followed by a 96 bit (12 byte) Salesforce generated IV, with the ciphertext following.

Use either third-party applications or the `decryptWithManagedIV` method to decrypt blobs encrypted with this method. Use the `encrypt` method if you want to generate your own initialization vector.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestEncryptWithManagedIV {

        public void testEncryptWithManagedIV(){
            String algorithmName = 'AES128';
            Blob key = Crypto.generateAesKey(128);
            Blob data = Blob.valueOf('Data to be encrypted');
            Blob encrypted = Crypto.encryptWithManagedIV(algorithmName, key, data);
            Blob decrypted = Crypto.decryptWithManagedIV(algorithmName, key, encrypted);
            String decryptedString = decrypted.toString();
            Assert.areEqual('Data to be encrypted', decryptedString, 'Error: the strings a

        }
    }
```

To invoke this method, run:

encryptWithManagedIV(algorithmName, secretKey, clearText, aaData)

Encrypts the *clearText* blob by using the specified algorithm and private key. Use this method when you want Salesforce to generate the initialization vector. This version of encryptWithManagedIV uses additional authentication data. CBC isn't supported for this method.

Signature

```
public static Blob encryptWithManagedIV(String algorithmName, Blob secretKey, Blob clearText, Blob aaData)
```

Parameters

algorithmName

Type: String

encryptWithManagedIV supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.

- `AES256-GCM`

secretKey

Type: Blob

Private key text. The length of *secretKey* must match the size required by *algorithmName*: 128 bits, 192 bits, or 256 bits, which is 16 bytes, 24 bytes, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key for you.

clearText

Type: Blob

The content you want to encrypt.

aaData

Type: Blob

Additional authentication data. This is required .

Return Value

Type: Blob

Contains the encrypted contents of *clearText*. For GCM, the return value contains the length of the IV (always 12) followed by a 96 bit (12 byte) Salesforce generated IV, with the ciphertext following.

Example

You can use your preferred Salesforce development environment to test this function. Create this Apex class.

```
public class TestEncryptWithManagedIV {

    public void testEncryptWithManagedIV(){
        String algorithmName = 'AES256-GCM';
        /*
           No IV if you specify AES256-GCM
        */
        Blob key = Crypto.generateAesKey(256);
        Blob data = Blob.valueOf('Data to be encrypted');
        Blob aad = Blob.valueOf('Additional tag');
        Blob encrypted = Crypto.encryptWithManagedIV(algorithmName, key, data, aad);
        Blob decrypted = Crypto.decryptWithManagedIV(algorithmName, key, encrypted, aa
        String decryptedString = decrypted.toString();
```

To invoke this method, run:

```
TestEncryptWithManagedIV teiv = new TestEncryptWithManagedIV();
teiv.testEncryptWithManagedIV();
```

generateAesKey(size)

Generates an Advanced Encryption Standard (AES) key.

Signature

```
public static Blob generateAesKey(Integer size)
```

Parameters

size

Type: Integer

The key's size in bits. Valid values are:

- 128
- 192
- 256

Return Value

Type: Blob

Contains the generated AES key.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class:

```
public class TestGenerateAESKey {

        public void testGenerateAESKey() {
          Blob key = Crypto.generateAesKey(128);
          System.debug('Generated AES Key: ');
          String strKey = EncodingUtil.base64Encode(key);
          System.debug(strKey);

        }
    }
```

To invoke this method, run the following:

```
TestGenerateAESKey tgaes = new TestGenerateAESKey();
tgaes.testGenerateAESKey();
```

generateDigest(algorithmName, input)

Computes a secure, one-way hash digest using the specified algorithm on the supplied *input* blob.

Signature

Type: String

The algorithm you want to use to generate the digest. Valid values for *algorithmName* are:

- MD5
- SHA1
- SHA3-256
- SHA3-384
- SHA3-512
- SHA-256
- SHA-512

input

Type: Blob

The content for which you want to generate the digest.

Return Value

Type: Blob

Contains the generated digest.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class:

```
public class TestGenerateDigest {

        public void testGenerateDigest(){
            Blob targetBlob = Blob.valueOf('ExampleMD5String');
            Blob hash = Crypto.generateDigest('MD5', targetBlob);
            String result = EncodingUtil.base64Encode(hash);
            System.debug('Value: ' + result);
        }
    }
```

To invoke this method, run the following:

```
TestGenerateDigest tgd = new TestGenerateDigest();
tgd.testGenerateDigest();
```

generateMac(algorithmName, input, privateKey)

Computes a message authentication code (MAC) for the *input* blob value using the private key and the specified algorithm.

Signature

```
public static Blob generateMac(String algorithmName, Blob input, Blob privateKey)
```

Parameters

algorithmName

Type: String

These are valid values for *algorithmName*.

input

Type: Blob

The content for which you want to generate the MAC.

privateKey

Type: Blob

The key to use to generate the MAC. You may supply a private key that has been encoded using Base64 encoding. However if you do, then you must also supply the Base64-encoded private key when verifying the MAC using the `verifyHMac` method. The value of *privateKey* can't exceed 4 KB.

Return Value

Type: Blob

The message authentication code.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class:

```
public class TestGenerateMAC {

        public void testGenerateMAC() {
            String salt = String.valueOf(Crypto.getRandomInteger());
            String key = 'key';
            Blob data = crypto.generateMac('HmacSHA256',
                                           Blob.valueOf(salt),
                                           Blob.valueOf(key));
            System.debug('Generated MAC: ');
            System.debug(EncodingUtil.base64Encode(data));
        }
    }
```

To invoke this method, run the following:

```
TestGenerateMAC tgm = new TestGenerateMAC();
tgm.testGenerateMAC();
```

getRandomInteger()

Returns a random integer value.

Signature

```
public static Integer getRandomInteger()
```

Return Value

Type: Integer

Returns a random 4-byte integer. Salesforce invokes the `java.security.SecureRandom` api to generate this number. For information on how the number is generated, see java.security.SecureRandom.

Example

```
public class TestGetRandomInteger {

        public void testGetRandomInteger() {
            Integer i1 = Crypto.getRandomInteger();
            Integer i2 = Crypto.getRandomInteger();
            System.debug('Integer 1: ' + i1);
            System.debug('Integer 2: ' + i2);
            Assert.areNotEqual(i1, i2, 'Sorry, those aren't random!');
            //This is just an example. This is not a true test of randomness
        }
    }
```

To invoke this method, run the following:

```
TestGetRandomInteger tri = new TestGetRandomInteger();
tri.testGetRandomInteger();
```

See Also

- java.security.SecureRandom

getRandomLong()

Returns a random long value.

Signature

```
public static Long getRandomLong()
```

Return Value

Type: Long

Returns a random 8-byte long. Salesforce invokes the `java.security.SecureRandom` api to generate this number. For information on how the number is generated, see java.security.SecureRandom.

Example

You can use your preferred Salesforce development environment to exercise this function. Create the following Apex class:

```
public class TestGetRandomLong {

        public void testGetRandomLong() {
            Long L1 = Crypto.getRandomLong();
            Long L2 = Crypto.getRandomLong();
            System.debug('Long 1: ' + L1);
            System.debug('Long 2: ' + L2);
            Assert.areNotEqual(L1, L2, 'Sorry, not random!');
            //This is just an example. This is not a true test of randomness
        }
    }
```

To invoke this method, run the following:

```
TestGetRandomLong trl = new TestGetRandomLong();
trl.testGetRandomLong();
```

sign(algorithmName, input, privateKey)

Computes a unique digital signature for the *input* blob value, using the specified algorithm and the supplied private key.

Signature

```
public static Blob sign(String algorithmName, Blob input, Blob privateKey)
```

Parameters

algorithmName

Type: String

input

signWithCertificate supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

Type: Blob

The data to sign.

privateKey

Type: Blob

The key to use for signing. The value of *privateKey* must be decoded using the `EncodingUtil base64Decode` method, and should be in RSA's PKCS #8 (1.2) Private-Key Information Syntax Standard form. The value can't exceed 4 KB.

Return Value

Type: Blob

The new digital signature.

Example

You can use your preferred Salesforce development environment to test this function. To run it correctly, you need a PKCS8 private key. At your terminal, use `openssl` to create one. First, create the key. Then convert it to PKCS8:

```
$ openssl genrsa -out myprivatekey.pem 1024
$ openssl pkey -in myprivatekey.pem -out myprivatekey.pkcs8.pem
```

After you create the PKCS8 compatible key, you decode just the key portion of the text (without the BEGIN PRIVATE KEY or END PRIVATE KEY lines) for the *privateKey* parameter.

```
public class TestSign {

        public void testSign() {
                Blob input = Blob.valueOf('Some text.');
                String algorithmName = 'RSA';
                String rawKey = '<text value of your pkcs8 private key>';

                //no BEGIN PRIVATE KEY or END PRIVATE KEY header/footer !
                Blob privateKey = EncodingUtil.base64Decode(rawKey);
                System.debug(privateKey);
                Blob signedKey = Crypto.sign(algorithmName, input, privateKey);
```

```
TestSign ts = new TestSign();
ts.testSign();
```

signWithCertificate(algorithmName, input, certDevName)

Computes a unique digital signature for the input blob value, using the specified algorithm and the supplied certificate and key pair.

Signature

```
public static Blob signWithCertificate(String algorithmName, Blob input, String certDevName)
```

Parameters

algorithmName

Type: String

signWithCertificate supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

input

Type: Blob

The data to sign.

certDevName

Type: String

The value listed in the `Unique Name` field for a certificate stored in the Salesforce org's Certificate and Key Management page to use for signing.
To access the Certificate and Key Management page from Setup, enter `Certificate and Key Management` in the **Quick Find** box, then select **Certificate and Key Management**.

Return Value

Type: Blob

The signed content.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class. For the *TestCertName* variable, use the unique name value for a self-signed or CA certificate that you have created in the org in which you run this test.

```
public class TestSignWithCert {

        public void testSignWithCert() {

            String algorithmName = 'RSA';
            Blob input = Blob.valueOf('Test Sign With Certificate.');
            String TestCertName = 'your-cert-unique-name';
            Blob signedKey = Crypto.signWithCertificate(algorithmName, input, TestCertName
        }
    }
```

```
tswc.testSignWithCert();
```

signXML(algorithmName, node, idAttributeName, certDevName)

Envelops the signature into an XML document.

Signature

```
public Void signXML(String algorithmName, Dom.XmlNode node, String idAttributeName, String certDevName)
```

Parameters

algorithmName

Type: String

signWithCertificate supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

node

Type: Dom.XmlNode

The XML node to sign and insert the signature into.

idAttributeName

Type: String

The full name (including the namespace) of the attribute on the node (XmlNode) to use as the reference ID. If `null`, this method uses the `ID` attribute on the node. If there's no `ID` attribute, Salesforce generates a new ID and adds it to the node.

certDevName

Type: String

The unique name for a certificate stored in the Salesforce org's Certificate and Key Management page to use for signing.
To access the Certificate and Key Management page from Setup, enter `Certificate and Key Management` in the **Quick Find** box, then select **Certificate and Key Management**.

Return Value

Type: void

This method doesn't return a value. The signature envelope is inserted within *node*.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class. For the *testCertName* variable, use the unique name value for a self-signed or CA certificate that you have created in the org in which you run this test.

```
public class TestSignXML {
  public void testSignXML() {
    String algorithmName = 'RSA';
    String testCertName = 'your-cert-unique-name';
    Dom.Document doc = new Dom.Document();
    String docToLoad = '<?xml version=\"1.0\"?>\n' +
    '<customers>\n' +
    '  <customer id="2">\n' +
    '    <name>Company One</name>\n' +
```

```
        //dump the content of the signed XML document to the debug log
        System.Debug(doc.toXmlString());
      }
    }
```

To invoke this method, run the following:

```
TestSignXML tswxml = new TestSignXML();
tswxml.testSignXML();
```

signXML(algorithmName, node, idAttributeName, certDevName, refChild)

Inserts the signature envelope before the specified child node.

Signature

`public static void signXml(String algorithmName, Dom.XmlNode node, String idAttributeName, String certDevName, Dom.XmlNode refChild)`

Parameters

algorithmName

Type: String

signWithCertificate supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

node

Type: Dom.XmlNode

The XML node to sign and insert the signature into.

idAttributeName

Type: String

The full name (including the namespace) of the attribute on the node (XmlNode) to use as the reference ID. If `null`, this method uses the `ID` attribute on the node. If there's no `ID` attribute, Salesforce generates a new ID and adds it to the node.

certDevName

Type: String

The unique name for a certificate stored in the Salesforce org's Certificate and Key Management page to use for signing.
To access the Certificate and Key Management page from Setup, enter `Certificate and Key Management` in the **Quick Find** box, then select **Certificate and Key Management**.

refChild

Dom.XmlNode

The XML node before which to insert the signature. If *refChild* is `null`, the signature is added at the end.

Return Value

Type: Void

following Apex class. For the *testCertName* variable, use the unique name value for a self-signed or CA certificate that you have created in the org in which you run this test.

```
public class TestSignXML_2 {
        public void testSignXML_2() {
            String algorithmName = 'RSA';
            String testCertName = 'your-cert-unique-name';
            Dom.Document doc = new Dom.Document();
            String docToLoad = '<?xml version="1.0"?>\n' +
            '<customers>\n' +
            '  <customer id="2">\n' +
            '    <name>Company One</name>\n' +
            '  </customer>\n' +
            '</customers>';
            doc.load(docToLoad);
            Dom.XmlNode rootNode = doc.getRootElement();
            Dom.XmlNode commentNode = rootNode.addCommentNode('SomeComment');

            System.Crypto.signXML(algorithmName, doc.getRootElement(), null, testCertName,

            //send the content of the signed XML document to the debug log
            System.Debug(doc.toXmlString());
        }

    }
```

To invoke this method, run the following:

```
TestSignXML_2 tswxml2 = new TestSignXML_2();
tswxml2.testSignXML_2();
```

verify(algorithmName, data, signature, publicKey)

Verifies the digital signature for the *data* blob using the specified algorithm and the supplied public key. Use this method to verify a blob signed by a digital signature created using a third-party application or the sign method.

Signature

```
public static Boolean verify(String algorithmName, Blob data, Blob signature, Blob publicKey)
```

Parameters

algorithmName

Type: String

verify supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

data

Type: Blob

The data to sign.

signature

Type:

Blob

The value of *publicKey* must be decoded using the `EncodingUtilbase64Decode` method, and be in X.509 standard format.

Return Value

Type: Boolean

`true` if and only if the signature is successfully verified.

Example

You can use your preferred Salesforce development environment to test this function. To run it correctly, you must:

- generate an X.509 private key and public certificate
- convert the private key to PKCS8
- extract the public key from the public certificate

You provide the private PKCS8 key to the `sign` method, and the extracted public key to the `verify` method (along with the signature you generate with `sign`.

At your terminal, use `openssl` to create the X.509 key pair:

```
$ openssl req -x509 -newkey rsa:2048 -keyout myPriv509.key -out myPub509.cert -days 365
```

Convert the private key to PKCS8:

```
openssl pkey -in myPriv509.key -out myPriv509pkcs8.pem
```

Extract the public key from `myPub509.cert`:

```
openssl x509 -in myPub509.cert -inform pem -pubkey -out myPub509.pem
```

After you create the `myPub509.pem` key, you decode just the key portions of the text (without the BEGIN PRIVATE KEY or END PRIVATE KEY lines) for both the *privateKey* and *publicKey* parameters.

```
public class TestVerify {
        public void testVerify() {

            String algorithmName = 'RSA';
            Blob input = Blob.valueOf('Here is some text.');

            //contents of myPriv509pkcs8.pem
            String myPriv509pkcs8 = 'contents of myPriv509pkcs8.pem';

            Blob privateKey = EncodingUtil.base64Decode(myPriv509pkcs8);

            Blob signature = Crypto.sign(algorithmName, input, privateKey);

            //contents of myPub509.pem
            String publicKeyTxt64 = 'contents of myPub509.pem';

            Blob publicKey = EncodingUtil.base64Decode(publicKeyTxt64);

            Boolean verified = false;
            verified = Crypto.verify(algorithmName, input, signature, publicKey);

            Assert.areEqual(true, verified);
```

```
TestVerify tv = new TestVerify();
tv.testVerify();
```

See Also

- X.509 Standard

verify(algorithmName, data, signature, certDevName)

Verifies the digital signature for the *data* blob using the specified algorithm and the public key associated with *certDevName*. Use this method to verify a blob signed by a digital signature created using a third-party application or the `signWithCertificate` method.

Signature

```
public static Boolean verify(String algorithmName, Blob data, Blob signature, String certDevName)
```

Parameters

algorithmName

Type: String

verify supports all these values for *algorithmName*. See Crypto Class for details on each algorithm.
`RSA`, `RSA-SHA1`, `RSA-SHA256`, `RSA-SHA384`, `RSA-SHA512`, `ECDSA-SHA256`, `ECDSA-SHA256-PLAIN`, `ECDSA-SHA384`, and `ECDSA-SHA512`

data

Type: Blob

The data to sign.

signature

Type:

Blob

The RSA or ECDSA signature.

certDevName

Type: String

The value listed in the `Unique Name` field for a certificate stored in the Salesforce organization's Certificate and Key Management page to use for signing.
To access the Certificate and Key Management page from Setup, enter `Certificate and Key Management` in the **Quick Find** box, then select **Certificate and Key Management**.

Return Value

Type: Boolean

Returns `true` if the signature is successfully verified.

Example

You can use your preferred Salesforce development environment to test this function. Create the following Apex class. For the *TestCertName* variable, use the unique name value for a self-signed or CA certificate that you have created in the org in which you run this test.

```
            String algorithmName = 'RSA';
            Blob input = Blob.valueOf('Test Sign With Certificate.');
            String TestCertName = 'your-cert-unique-name';
            Blob signedKey = Crypto.signWithCertificate(algorithmName, input, TestCertName);

            Boolean verified = false;
            verified = Crypto.verify(algorithmName, input, signedKey, TestCertName);
            Assert.areEqual(true, verified);
        }
    }
```

To invoke this method, run the following:

```
TestVerify_2 tv_2 = new TestVerify_2();
tv_2.testVerify_2();
```

verifyHMac(algorithmName, data, privateKey, macToVerify)

Verifies the HMAC signature for the *data* blob using the specified algorithm, input data, private key, and the mac. Use this method to verify a blob signed by a digital signature created using a third-party application or the sign method.

Signature

```
public static Boolean verifyHMac(String algorithmName, Blob data, Blob privateKey, Blob macToVerify)
```

Parameters

algorithmName

Type: String

These are valid values for *algorithmName*.

- `hmacMD5`
- `hmacSHA1`
- `hmacSHA256`
- `hmacSHA512`

data

Type: Blob

The data to sign.

privateKey

Type: Blob

If the private key used to generate the MAC was Base64 encoded, then the value of *privateKey* must also be Base64 encoded. The value cannot exceed 4 KB.

hmacToVerify

Type: Blob

The value of the mac must be verified against the provided *privateKey*, *data*, and *algorithmName*.

Return Value

Type: Boolean

The verification status of the data to verify.

```
public class TestVerifyMAC {

        public void testVerifyMAC() {
            String salt = String.valueOf(Crypto.getRandomInteger());
            String key = 'key';
            Blob data = crypto.generateMac('HmacSHA256',
                Blob.valueOf(salt),
                Blob.valueOf(key));
            System.debug('Generated MAC: ');
            System.debug(EncodingUtil.base64Encode(data));

            Boolean verified = false;

            verified = Crypto.verifyHMac('HmacSHA256', Blob.valueOf(salt), Blob.valueOf(ke
            Assert.areEqual(true, verified);
        }
    }
```

To invoke this method, run the following:

```
TestVerifyMAC tvm = new TestVerifyMAC();
tvm.testVerifyMAC();
```

DID THIS ARTICLE SOLVE YOUR ISSUE?
Let us know so we can improve!

Share your feedback