Chapter 3. HASH FUNCTIONS AND MAC CODES IN .NET

This section presents the hash functions and their immediate derivative *Message Authentication Codes (MAC)* that are supported by the .NET framework. We also make use of random number generators.

The .NET framework supports the now deprecated but still largely used MD5 (128 bit) and SHA1 (160 bit). Besides these, there is also support for the (soon to be replaced) current standard SHA2 in all three output sizes 256, 384 and 512 bit and the less frequent RIPEMD (160 bit).

MACs (Message Authentication Codes) are also named keyed hash functions since they are built from a hash function with the use of a secret key. But there are also exceptions to this rule and it happens for MAC codes to be built from symmetric encryption functions rather than hash functions. The .NET framework contains one such exception which is the *MACTripleDES*, a MAC code build on 3DES. The other MAC code that is supported by .NET is HMAC, which is indeed a keyed hash function and the preferred alternative, it can be built on any of the hash functions available in the framework: MD5, SHA1, SHA2 or RIPEMD. Figure 1 shows the class organization for hash algorithms and MAC codes, we can see again the distinction between abstract and concrete classes.

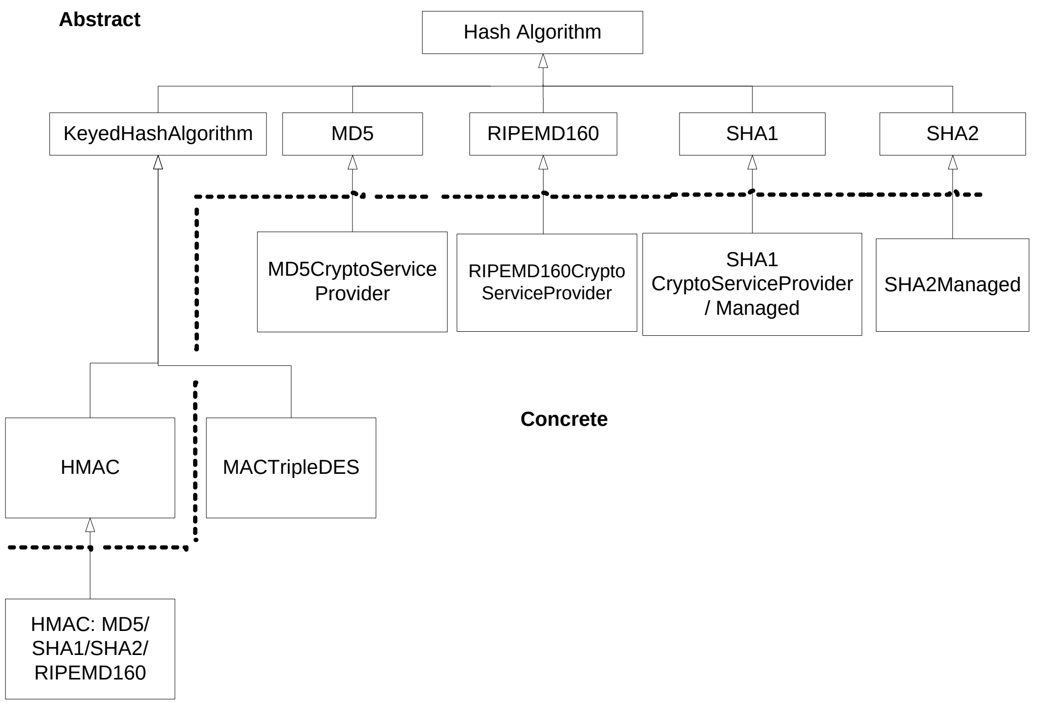

Abstract
Hash Algorithm
KeyedHashAlgorithm
MD5
RIPEMD160
SHA1
SHA2
MD5CryptoService Provider
RIPEMD160Crypto ServiceProvider
SHA1 CryptoServiceProvider / Managed
SHA2Managed
Concrete
HMAC
MACTripleDES
HMAC: MD5/ SHA1/SHA2/ RIPEMD160

Figure 1. Hash functions and keyed hash functions in .NET

3.1 HASH FUNCTIONS

Hash functions are derived from the abstract class *HashAlgorithm* located in the aforementioned *System.Security.Cryptography* namespace. Some properties and methods are outlined in Tables 1 and 2.

	Get/Set	***Type***	**Brief Description**
InputBlockSize	*g*	*Int*	Bit size of the input block, returns 1 unless overwritten
OutputBlockSize	*g*	*Int*	Bit size of the output block, returns 1 unless overwritten
HashSize	*g*	*Int*	Bit size of the hash
Hash	*g*	*Byte[]*	Value of the hash

Table 1. Some properties for hash functions in .NET

	Return type	**Brief Description**
Create()	*HashAlgorithm*	Creates the object (SHA1 is the default instance)
Create(String)	*HashAlgorithm*	Creates the object with the string specifying the name of the particular implementation given as string (MD5, SHA1, etc.)
ComputeHash(Byte[])	*Byte[]*	Computes the hash from a byte array
ComputeHash(Stream)	*Byte[]*	Computes the hash from a stream object
ComputeHash(Byte[], Int32, Int32)	*Byte[]*	Computes the hash from a specific region of a byte array
TransformBlock(byte[] inputBuffer, int inputOffset, int inputCount, byte[] outputBuffer, int outputOffset)	*Int*	Computes the hash of a specified region of a byte array and copies the region to the specified region of the output byte array. Return the number of bytes written.
TransformFinalBlock(byte[] inputBuffer, int inputOffset, int inputCount)	*Byte[]*	Computes the hash of a specified region of a byte array, returns a copy a of the part of the input that is hashed

Table 2. Some methods for hash functions in .NET

To compute the hash of a byte array or stream you can simply call the *ComputeHash* method as outlined in Table 3. In this example we also used a *RandomNumberGenerator* object to generate some arbitrary values that are later hashed. To generate random values, you simply have to create a *RandomNumberGenerator* object and make a call to the *GetBytes* method on a specific byte array.

```
MD5CryptoServiceProvider myMD5 = new MD5CryptoServiceProvider();
RandomNumberGenerator rnd = RandomNumberGenerator.Create();
byte[] input = new byte[20];
byte[] hashValue;
//generates some random input
rnd.GetBytes(input);
//computes the hash
hashValue = myMD5.ComputeHash(input);
```

Table 3. Example for generating some random bytes and computing their hash

In Table 4 we then show how to compute the hash of a given file, this is a frequently used procedure to check the integrity of files, or to compare if two files (or objects) are the same, as only identical objects can hash to the same value (assuming the hash function is collision free).

```
FileStream fileStream = new FileStream("C:\\TEMP\\x.pdf",
FileMode.Open);
fileStream.Position = 0;
hashValue = myMD5.ComputeHash(fileStream);
```

Table 4. Example for computing the hash of a given file

3.2 Keyed Hash Functions

The .NET framework provides an implementation for the HMAC keyed hash function. The properties and methods for *KeyedHashAlgorithm* objects are almost identical to that of *HashAlgorithm* (a class which they do inherit). The only additional property, is the one to get or set the key as outlined in Table 5.

	Get/Set	***Type***	**Brief Description**
Key	*g/s*	*Byte[]*	Value of the key for the HMAC

Table 5. The *Key* property of keyed hash algorithms in .NET

In Tables 6 and 7 we show how to instantiate a HMAC with a particular hash function, how to generate the authentication tag with *ComputeMAC(byte[] mes, byte[] key)* and then verify it with *CheckAuthenticity(byte[] mes, byte[] mac, byte[] key)*.

```
private HMAC myMAC;

public MACHandler(string name)
{
if (name.CompareTo("SHA1") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA1(); }
if (name.CompareTo("MD5") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACM
                                     D5(); }
if (name.CompareTo("RIPEMD") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACR
                                     IPEMD160(); }
if (name.CompareTo("SHA256") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA256(); }
if (name.CompareTo("SHA384") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA384(); }
if (name.CompareTo("SHA512") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA512(); }
}
```

Table 6. Creating a HMAC object with a particular hash function

```
public bool CheckAuthenticity(byte[] mes, byte[] mac, byte[] key)
{
  myMAC.Key = key;
  if (CompareByteArrays(myMAC.ComputeHash(mes), mac, myMAC.HashSize /
8) == true)
   {
      return true;
   }
  else
   {
      return false;
   }
}

public byte[] ComputeMAC(byte[] mes, byte[] key)
{
  myMAC.Key = key;
  return myMAC.ComputeHash(mes);
}

public int MACByteLength()
{
  return myMAC.HashSize / 8;
}

private bool CompareByteArrays(byte[] a, byte[] b, int len)
{
  for (int i = 0; i < len; i++)
    if (a[i] != b[i]) return false;
  return true;
}
```

Table 7. Computing the HMAC and then verifying the authenticity of a message

3.3 Hash Functions and MAC Codes as CryptoStreams

A final trick that may be useful to know is that you can pass hash functions or HMACs as transformations embedded into *CryptoStreams*. In Table 8 we show such an example. The streams that we use will not store the data that is written into them, i.e., they receive Stream.Null at initialization. In order to retrieve the hash or HMAC value we then simply call the Hash property of the cryptographic objects, i.e., *hmac.Hash* and *hash.Hash*.

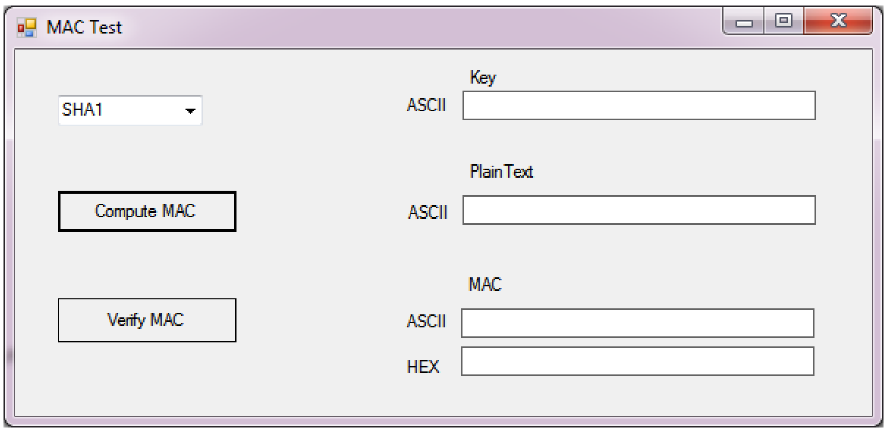
```
RandomNumberGenerator rnd = RandomNumberGenerator.Create();
byte[] key = new byte[16];
rnd.GetBytes(key);
byte[] input = new byte[20];
rnd.GetBytes(input);

HMACSHA256 hmac = new HMACSHA256(key);
SHA256Managed hash = new SHA256Managed();
```

```
CryptoStream cs_hmac = new CryptoStream(Stream.Null, hmac,
CryptoStreamMode.Write);
CryptoStream cs_hash = new CryptoStream(Stream.Null, hash,
CryptoStreamMode.Write);

cs_hmac.Write(input, 0, input.Length);
cs_hmac.Close();

cs_hash.Write(input, 0, input.Length);
cs_hash.Close();
```

Table 8. Example for HMACSHA256 and SHA256 used in *CryptoStreams*

3.4 Exercises

2. Write a C# application that allows a user to select a Hash or HMAC algorithm from a Combo Box, generate keys (in case of HMAC), hash messages and verify (in case of HMAC) their hashes. Display the plain text and hash both in ASCII and HEX; also display the time required by the hash and HMAC operations. A suggestion for starting the interface is below, but feel free to modify it at will. Results should be presented in a tabular form as shown below.



	SHA1 (CSP)	SHA1 (Managed)	SHA256 (CSP)	SHA256 (Managed)	SHA384 (CSP)	SHA256 (Managed)	SHA512 (CSP)	SHA512 (Managed)	MD5 (CSP)	RIPEMD (Managed)
seconds/block										
bytes/second (from RAM)										
bytes/second (from HDD)										

Table 9. Computational cost for hash functions

2. Write a program that searches, by generating random values, for hashes that have all the last k bits set to 0 (k is given as parameter by the user). Give an estimation to find such values for a given k.

Remark. You can recycle some of the code below for the interface of exercise 1.

```
private void buttonCompute_Click(object sender, EventArgs e)
{
  MACHandler mh = new MACHandler(comboBoxMAC.Text);
  byte[] mac =
              mh.ComputeMAC(myConverter.StringToByteArray(textBoxPlain.
              Text),myConverter.StringToByteArray(textBoxKey.Text));
  textBoxMAC.Text = myConverter.ByteArrayToString(mac);
  textBoxMACHEX.Text = myConverter.ByteArrayToHexString(mac);
}


private void buttonVerify_Click(object sender, EventArgs e)
{
   MACHandler mh = new MACHandler(comboBoxMAC.Text);
```

```
    if
                (mh.CheckAuthenticity(myConverter.StringToByteArray(textB
                oxPlain.Text),
                myConverter.HexStringToByteArray(textBoxMACHEX.Text),myCo
                nverter.StringToByteArray(textBoxKey.Text)) == true)
    {
       System.Windows.Forms.MessageBox.Show("MAC OK !!!");
    }
    else
    {
       System.Windows.Forms.MessageBox.Show("MAC NOT OK !!!");
    }
}
```