Chapter 5. THE DSA SIGNATURE ALGORITHM IN .NET

This section presents the DSA (Digital Signature Algorithm) as implemented in .NET. The .NET framework contains two digital signature schemes, the RSA which was previously discussed and DSA, also known as DSS (Digital Signature Standard). The DSA is a discrete logarithm based signature scheme, based on the ElGamal signature, which was standardized by NIST.

5.1 BRIEF THEORETICAL BACKGROUND

You are referred to the lecture material for more details on the DSA. However, to make things clearer, we do a brief recap on how DSA works. The details of this algorithm are more complicated than for the RSA, in particular the details are somewhat uneasy to memorize as the construction appears less natural than the RSA. The straight-forward idea of using the encryption key for verification and the decryption key for signing does not work anymore, however, the algorithm is in fact slightly more efficient and secure than the RSA and not hard to implement (it is based on the same core operation: modular exponentiation).

How the DSA signature works. As any public key signature, the DSA is a collection of three algorithms:

- ***Key generation:*** generate a random prime p and a second random prime q such that it divides $p-1$ then select an element g of Z_p of order q and a random number a from Z_p. The public key is $Pb = (g, g^a \bmod p, p)$ and the private key is $Pv = (g, a, p)$ **(q is fixed at 160 for the .NET implementation due to the use of SHA1)**
- ***Signing***: given the message m, use the hash function (SHA1 in .NET) to compute the hash of the message h, then select a random $k \in (0, p-1)$ and compute $r = g^k \bmod p$ then $s = k^{-1}(\mathrm{h} + \mathrm{ar}) \bmod (p-1)$. The signature is the pair (r, s).
- ***Verification***: given the signature (r, s), first check that $r \in (0, p)$, $s \in (0, q)$ (if not the signature is considered false) otherwise verify that $v = r$ where $v = g^{u_1} y^{u_2}, u_1 = wh \bmod q, u_2 = rw \bmod q, w = s^{-1}$ and return true if this holds (otherwise the signature is considered false).

Fortunately, you do not need to remember all these details in order to use this signature scheme in .NET, all you have to do is to call the methods for signing and verification. We discuss these next.

5.2 DSACryptoServiceProvider: Properties and Methods

The DSA implementation in .NET supports keys from 512 to 1024 bits in 64 bit increments. The key size can be specified via the constructor of the *DSACryptoServiceProvider* class which will generate a random DSA key. Similar to the case of the RSA, the constructor also allows initialization with an existing key given as *CspParameters* object. However, the methods for signing and verification do not offer the possibility of using an external hash object, in .NET this signature is bound to SHA1. These methods are summarized in Table 1, the distinction with the RSA is the absence of the hash algorithm as parameter since this is implicitly set to SHA1. The *DSACryptoServiceProvider* also has an additional *VerifySignature* method that takes the hash and signature of the message as input.

	Return type	**Brief Description**
ExportParameters (bool includePrivateParameters)	*DSAParameters*	Gets the DSA key as RSAParameters object. The Boolean specifies if the private part of the key is or not included.
ImportParameters (DSAParameters parameters)	*void*	Sets the DSA key from DSAParameters object
ToXmlString (bool includePrivateParameters)	*string*	Gets the DSA key as string in XML format. The Boolean specifies if the private part of the key is or not included.
FromXmlString (bool includePrivateParameters)	*void*	Sets the DSA key from a string in XML format.
SignData(byte[] buffer)	*byte[]*	Signs the given array of bytes with the specified hash algorithm, returns the signature as array of bytes

SignData(Stream inputStream)	*byte[]*	Same as previously, but this time the data is given as stream
SignData(byte[] buffer, int offset, int count)	*byte[]*	Signs the byte array starting from *offset* for *count* bytes
SignHash(byte[] hash, string str)	*byte[]*	Signs the hash of the data, the string is the name of the algorithm that was used to hash the data
VerifyData(byte[] buffer, byte[] signature)	*bool*	Verifies the signature given a hash algorithm as object, the signature and message as byte arrays
VerifyHash (byte[] Hash, string str, byte[] Signature)	*bool*	Verifies the signature given the hash of the message and the name of the hash algorithm
VerifySignature(byte[] Hash, byte[] Signature)	*bool*	Verifies the signature given the hash of the message

Table 1. Methods from the *DSACryptoServiceProvider* class

Signing with DSA in .NET. Signing requires the instantiation of a *DSACryptoServiceProvider* object and then calling one of the signing methods, same for verification. This is suggested in the code from Table 2.

```
DSACryptoServiceProvider myDSA = new DSACryptoServiceProvider(512);
byte[] sig = myDSA.SignData(data);
bool verify = myDSA.VerifyData(data, sig);
```

Table 2. Example for signing a byte array and verifying the signature in .NET with DSA

5.3 The Structure of the Public and Private Key

We now enumerate the parameters of the DSA private and public key:

- ✓ **P** – the prime that defines the group, i.e., p,
- ✓ **Q** – the factor of p-1 which gives the order of the subgroup, i.e., q, (this is always 160 bits in .NET)
- ✓ **G** – the generator of the group, i.e., g,
- ✓ **Y** – the value of the generator to X, i.e., $y = g^x \bmod p$
- ✓ **J** – a parameter specifying the quotient from dividing p-1 to q, i.e., $j = (p - 1)/q$,
- ✓ **Seed** – specifies the seed used for parameter generation,
- ✓ **Counter** – a counter value that results from the parameter generation process,
- ✓ **X** – a random integer, this is the secret part of the key, i.e., parameter a from the description of the scheme.

Exporting and importing keys as XML strings. Similar to the case of RSA the key can be exported to (or imported from) XML Strings. In Tables 3 and 4 we show a DSA key exported from .NET with and without the private part.

<DSAKeyValue>
<P>
sRp/2qfasQ+6ObB/6+7HqyZnmgp0drn7G/ewLihzFfiJrVS15Wu5slPXYY8lIpiqbwgVWj5UMoV1ynnmx392YQ==
</P>
<Q>
+DqDOhkIdeiQrtipZf6d/ei35Yc=
</Q>
<G>
NElPMJMiLsqzHWyFmQeLNESbdmRNTta78aApURYyCqZ9CVTQCZTwX/N5YpulkCKGKwOxMkXRdfAB0XVDQj/nJQ==
</G>
<Y>
KYtWDqa9aRI/bP5q82sfpSutSWJqDnkS9INGhZbdHxHcJw4XMU/ihIHUzS3zkODneMnj3kz0Ly3jMJvkcm15kw==
</Y>
<J>
tqXwIvpqvwSLuUWWfcrGaUyl9AP07V0qfib1UtBD2xyf0c9sHacjniyQbqA=

```
</J>
<Seed>
nVHH51WY6AQqRGBXYDg+zQnHF5s=
</Seed>
<PgenCounter>
Ag==
</PgenCounter>
<X>
NhAM2W6d+VISPUZ967Zfc8v8D+8=
</X>
</DSAKeyValue>
```

Table 3. DSA key exported as XML string with private parameters

```
<DSAKeyValue>
<P>
sRp/2qfasQ+6ObB/6+7HqyZnmgp0drn7G/ewLihzFfiJrVS15Wu5slPXYY8lIpiqbwgVWj
5UMoV1ynnmx392YQ==
</P>
<Q>
+DqDOhkIdeiQrtipZf6d/ei35Yc=
</Q>
<G>
NElPMJMiLsqzHWyFmQeLNESbdmRNTta78aApURYyCqZ9CVTQCZTwX/N5YpulkCKG
KwOxMkXRdfAB0XVDQj/nJQ==
</G>
<Y>
KYtWDqa9aRI/bP5q82sfpSutSWJqDnkS9INGhZbdHxHcJw4XMU/ihIHUzS3zkODneM
nj3kz0Ly3jMJvkcm15kw==
</Y>
<J>
tqXwIvpqvwSLuUWWfcrGaUyl9AP07V0qfib1UtBD2xyf0c9sHacjniyQbqA=
</J>
<Seed>
nVHH51WY6AQqRGBXYDg+zQnHF5s=
</Seed>
<PgenCounter>
Ag==
</PgenCounter>
</DSAKeyValue>
```

Table 4. DSA key exported as XML string without private parameters (just the public key)

Exporting and importing keys as byte arrays. Identical to the case of RSA, keys can be imported and exported as *System.Security.Cryptography.DSAParameters* which is a structure containing a byte array for each of the previously described parameters. This is suggested in Figure 1.

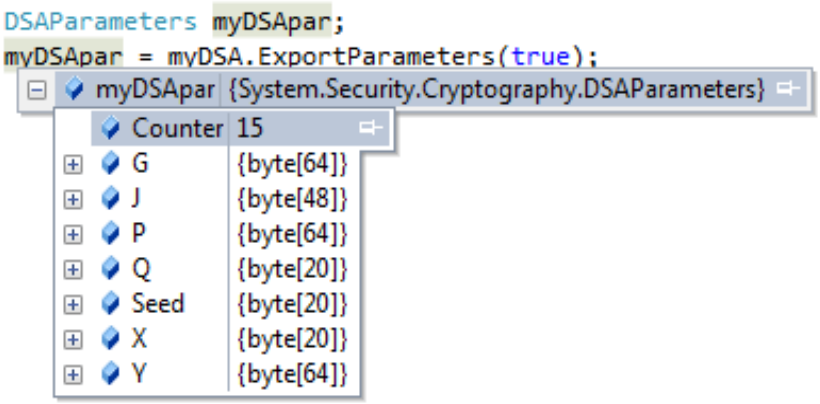

Figure 1. Fields of the *DSAParameters* structure

5.4 EXERCISES

2. Evaluate the computational cost of DSA signature in .NET in terms of: key generation, signing and verification time. Results have to be presented in a tabular form as shown below.

512 bit	640 bit	768 bit	1024 bit

Table 5. Cost of DSA key generation

512 bit	640 bit	768 bit	1024 bit

Table 6. Cost of DSA signing

512 bit	640 bit	768 bit	1024 bit

Table 7. Cost of DSA verification