Introduction To Web Security

Logica – Java SIG – May 24th, 2012

Bernard Jorion

Latest Version: October 6, 2012

Contents

1. Introduction

The topic of this document is to present the main methods to secure communications on internet.

This presentation is technical, but not too much. Only a simple mathematical background is necessary to follow it, and you do not even need any Java knowledge.

At the end of this document, you will be familiar with the digital certificates, and the tools to create and analyse them.

2. The Security Issue

The main protocol used on the web is the **HTTP** protocol (Hypertext Transfer Protocol). It is simple and easy to use, but does not make any guarantees concerning the security of the data.

Security must be understood here in a broad sense: anybody could easily listen to your communications without your knowledge, and even modify it on the run. Most of the time, this will not be an issue: if you consult e.g. your favourite weather website, the information is public anyway and nobody will want to modify it before it reaches your computer.

On the other hand, if you need to send or receive sensitive information (like reading your mails, buying on-line), you cannot afford not to secure your data.

The issue is actually twofold: first, you need to encrypt your data using an ***encryption algorithm***. Then you also need to identify yourself. You do that using ***digital certificates***.

When you combine an encryption algorithm and the digital certificates to the HTTP protocol, you obtain the **HTTPS** protocol (S is for Secure).

The best news of all are that you do not need a PhD in math to secure your data, the servers and the browsers take care of almost all the work.

3. The Symmetrical Algorithms

First of all, you need to encrypt your data using an encryption algorithm. The “perfect” way to encrypt a piece of information is to use a ***symmetrical algorithm*** with a key as long as the message.

A **key** is a piece of information that allows you to encrypt the data. For instance, if you want to encrypt the name **LOGICA**, you could decide to replace each letter by the next one in the alphabet. You will obtain **KPHJDB** since K = L + 1, P = O + 1, etc... In this case, the algorithm consists in replacing each letter by another one, and the key is the “distance” between the new and the original letter (here: + 1). Such an algorithm is called symmetrical because once you have the key and the encrypted message, you can easily retrieve the original message (you take the previous letter in the alphabet).

Such an algorithm does not seem very secure, but let’s say that for each letter in the name **LOGICA**, we choose a different distance between the original letter and the new one. For example, let’s use the distances [+5, +10, +7, +9, -2, +2]. We will obtain the word **QYNRAC** (since Q = L + 5, Y = O + 10, etc...). Here the key is the list of all the distances, and is *as long as the message* (6 elements).

It is simply impossible to decrypt the encoded message if you do not know the complete key. But this is only true if a given key is never used more than one (otherwise statistical methods could provide information on the original message).

The symmetrical algorithms have two major drawbacks: the first one is that you need to agree on a key with your correspondent (before sending the message), and the second one is that key must be as long as the message – so if you want to encrypt a 1 megabyte picture, you will need a 1 megabyte key.

The second drawback can be solved by using shorter keys. You will indeed lose some security, but if a thief still needs 10.000 years to decode your message, that’s something you can live with ☺.

Still, the first drawback is a blocking point. On internet, you may want to exchange information with somebody you do not know on the other side of the world (a client in China...) and there is simply no way to agree beforehand on a key (obviously you cannot transmit the key insecurely on the web, that would defeat the whole encryption process).

To solve that issue, I would like to present you the ***asymmetrical algorithms***.

4. The RSA Asymmetrical Algorithm

With an asymmetrical algorithm, what you encrypt with one key cannot be decrypted with the *same* key. In other words, it works one way only (while the key of a symmetrical algorithm works both ways).

An asymmetrical algorithm uses two keys: a **private key** and a **public key**. If somebody sends me a message encrypted with my *public* key (which, by definition, is openly accessible to everyone), only I will be able to decrypt it using my *private* key.

I will present here the most used asymmetrical algorithm: the RSA algorithm.

Note: I present the algorithm only for the sake of completeness, if you want to use HTTPS there is absolutely no need to know anything about the inner workings of such an algorithm. You can view it as a black box.

The RSA algorithm was developed in the years '70 by three scientists: *Rivest, Shamir* and *Adleman* (whence the name RSA).

Let's say A and B are prime numbers. We define

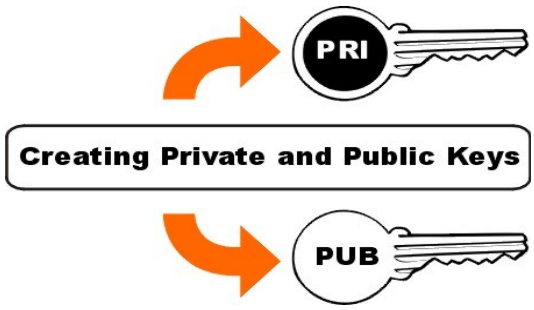
$$P = A * B.$$

Computing P (**multiplying** A per B) is very simple. But the mathematical properties of prime numbers are such that there are no way of **factorizing** P (retrieving the original A and B), except by brute force.

$$P => A ?,\ B ?$$

(Well, if P is 15, you can guess that A and B are 3 and 5, but if P is a thousand-digit long, good luck to retrieve A and B).

Knowing that, you can create a **private key** base on A and B, and a ***public key*** based on P (= A * B).



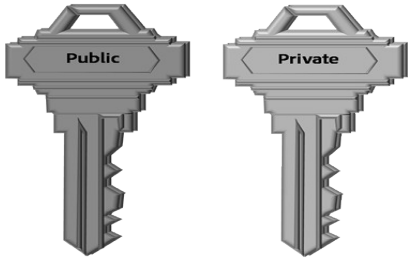
Note: the actual creation of the private and the public keys is much more complicated that just taking A,B on one side, and P on the other side. But let's keep thing simple. For the curious, you can find more information on the net.

The principle of asymmetric algorithms is that you publish your **public key**, and you keep the **private key** safe on your computer.

Mathematical properties of the asymmetrical algorithms are such that:

1. A message signed (encrypted) with my **public key** can only be decrypted with my **private key**, - so I am sure nobody else will be able to read the data

2. A message signed my with **private key** can only be decrypted with my **public key** – in this case anybody will be able to read the message I send, but only I could have send it, and not somebody else pretending to be me.



Asymmetrical algorithms have their own drawbacks: first, the keys (public or private) need to be much longer that symmetrical keys for the same level of security, then signing (encrypting) a message takes a lot of CPU power (due to the fact the keys are longer).

So how do you get the best of both worlds? The security of asymmetric algorithms with the efficiency of the symmetric ones? Stay tuned...

5. Combining Symmetrical and Asymmetrical Algorithms

Since asymmetrical algorithms are far less efficient that symmetrical ones, when the server wants to establish a secure connection with a browser (using HTTPS), it follows the steps:

- The server uses an *asymmetrical* algorithm (normally RSA) to connect to the browser on the client side.
- Once the connection is established, it sends to the browser a newly generated **symmetrical key** (only valid for this transaction)
- The server stops using the asymmetrical algorithm and starts a new connection encrypted with a *symmetrical* algorithm, using the symmetrical key just previously exchanged.

Once again, you do not need to know any mathematical details of any algorithm, the server and the browser will automatically encrypt and decrypt the data for you.

6. Digital Certificates

We still have one important issue to solve. Let's say I publish my public key somewhere on the net, so that you can use it to send me encrypted message. How can you be sure it is really my personal public key, and not from someone pretending to be me? You do not want to send me personal information in an encrypted message, just to realise only a thief could read it.

Sure I could encrypt my public key, but how would you decrypt it? That's the egg-and-chicken problem.

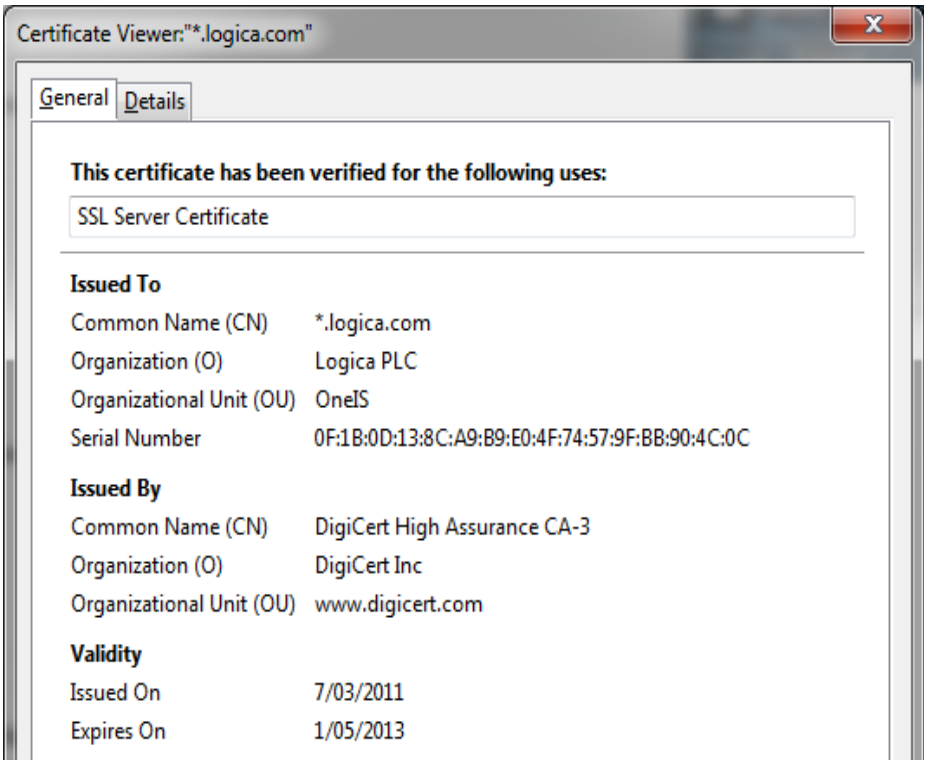
Welcome to the world of ***digital certificates***. A certificate is the equivalent of the electronic ID card, proving you are who you claim to be. Such a certificate is issued by a well-known company that has taken care to check your real identity (like the State that delivers a classical ID).

Such a certificate contains your name, but most importantly your email and the address of your website (both guaranteed to be unique).

Such a company is called a **Certification Authority**. Famous examples are **VeriSign** and **GoDaddy**.

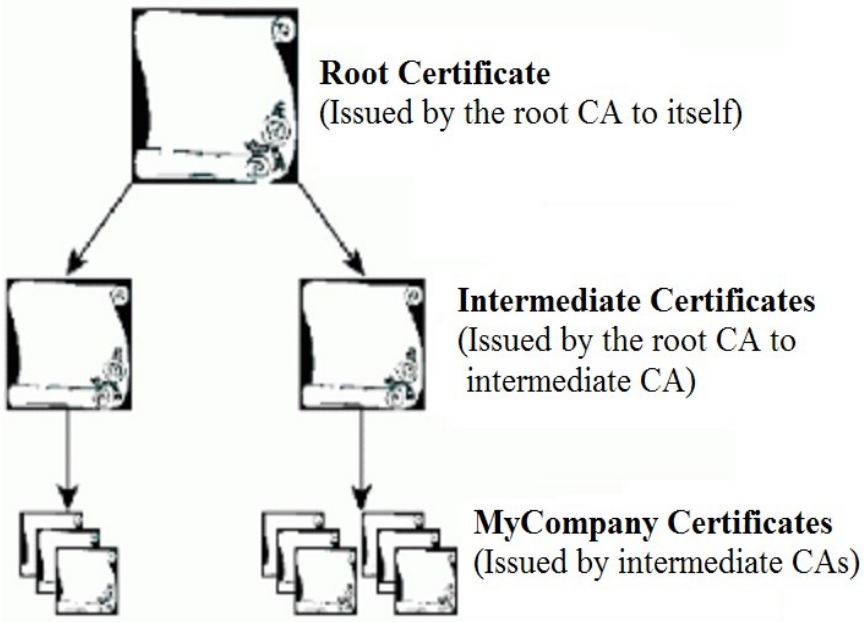
The most common format of a certificate is named **X.509** (that's a strange name, but it does not matter).

For example, if you go on https://beweblinks.logica.com/, by clicking on the lock icon in the URL bar, you can view the certificate of Logica:

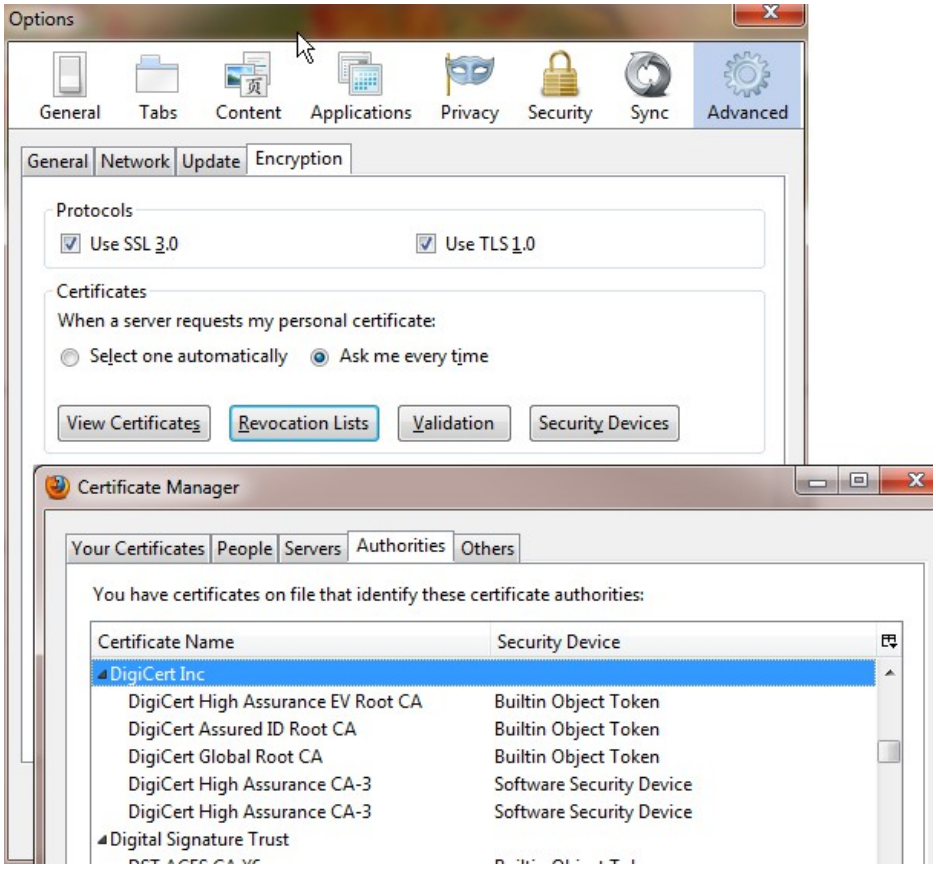


The information displayed include the web site (*.logica.com), organization (Logica PLC), validity period (from 7/3/2011 to 1/5/2013), the name of the issuer (`DigiCert High`), and so on...

The browser knows it can trust your certificate because it relates to another certificate, which itself relates to another certificate, and so on up to a **root certificate** which is the Certification Authority's own certificate.



Every browser owns a list of trusted root certificates. In Firefox, it is visible in `Options > Advanced > Encryption > View Certificates`:



You can see above that DigiCert Inc is a certification authority recognized by the browser.

This mechanism is known as a **chain of trust**.

The following steps are necessary to generate a valid certificate:

1. You create manually your own certificate (with information about you),
2. You ask a *Certification Authority* (C.A.) to validate it (they check one way or another you are who you claim to be, for example they can contact you by phone)
3. Once they are satisfied, they sign your certificate using one of their own (normally, not the “root” one, but an intermediate)
4. You can now install your certificate on your server
5. The browser, when establishing a secure connection with your server, will receive your certificate.
6. The first time the browser connects, it does not know you yet, so it will check the certificate by retrieving its parent, and the parent of its parent, and so on until the moment it can connect to a trusted certificate in its own list.

As you can guess, certificates are rather expensive and are valid for a limited time only.

7. The HTTPS Protocol

Now we have seen every piece of the puzzle. Let's put everything together.

- The **SSL protocol** (Secure Socket Layer) is a cryptographic protocol that provides security over Internet
- SSL uses the **RSA** asymmetrical algorithm and the **X.509 digital certificates**.
- The SSL protocol 3.0 was renamed TLS protocol 1.0 (Transport Layer Security) in 2001.
- **HTTPS = HTTP + SSL**

8. How Secure is RSA?

Length of P = A * B	Secure until
1024 bits (300 digits)	2010
2048 bits (600 digits)	2030
3072 bits (900 digits)	?

The current standard length of a RSA key is **2048** bits. Keep in mind that with every doubling of the RSA key length, encryption and decryption time is 6-7 times slower.

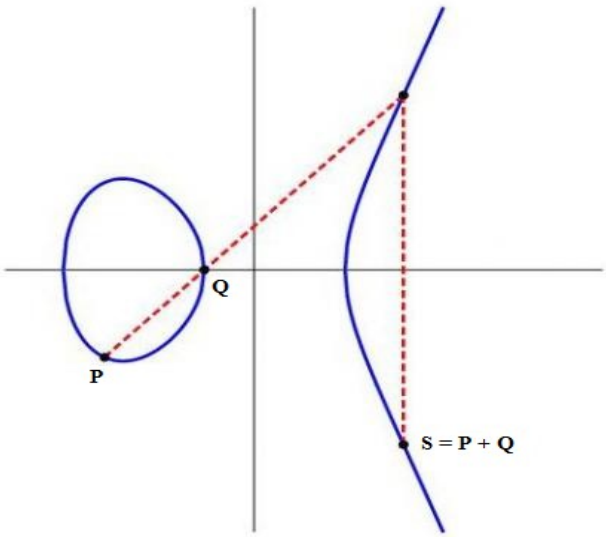
9. The Elliptic Curve Algorithm

Other algorithms, more efficient than the RSA, have been developed since the '70. One already implemented in the latest versions of the JDK is the ***Elliptic Curve Algorithm***. Once again, you do not need to understand the mathematical details to use it, but it is always interesting to know what you are using.

An elliptic curve follows the equation:

$$y^2 = x^3 + ax + b$$

The curve looks like:



The points P and Q belong to the curve. Mathematical properties of the curve are such that computing **S = P + Q** is easy, while retrieving **P or Q** from **S** is very difficult (like factorizing P = A * B).

The main advantage of such an algorithm is that, for the same level of security, the length of the Elliptic Curve key is much shorter than the RSA key, whence a short computation time.

Symmetric key length (bits)	RSA key length (bits)	Elliptic Curve Key length (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Source: http://www.nsa.gov/business/programs/elliptic_curve.shtml

10. Generation of Certificates & PKI Tools

10.1. PKI

The term **PKI** (Public Key Infrastructure) is used to refer to the set of hardware, software, people, policies and procedures needed to create, manage, distribute, use, store and revoke **digital certificates**.

In cryptography, a PKI is an arrangement that binds public keys with respective user identities by means of a certificate.

10.2. Java KeyTool

KeyTool is a command-line utility from the JDK used to manage a **keystore** (database) of cryptographic keys and X.509 digital certificates.

The following command lets you generate a *key pair* (public and private key) and a self-signed certificate. A self-signed certificate is signed by yourself and has of course no value (no browser will accept it), but it is the first step to obtain a "true" certificate, signed by a C.A.

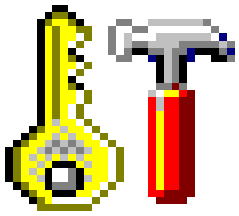
```
keytool
 -genkeypair
 -keystore mykeys.jks
 -validy 1000
 -dname "CN=John Smith, O=Logica, L=Brussels, C=Belgium"
```

genkeypair is here the main command, everything else is optional. The **validity** is the number of days you certificate will be considered as valid (by default 90 days), **keystore** gives the name of the file created by keytool (under a format specific to Java) and most important, **dname** is "distinguished name" of the user.

Using `keytool -list -keystore mykeys.jks`, you will be able to retrieve the stored information in the keystore.

You will find more information on the numerous command on
http://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html

10.3. Portecle

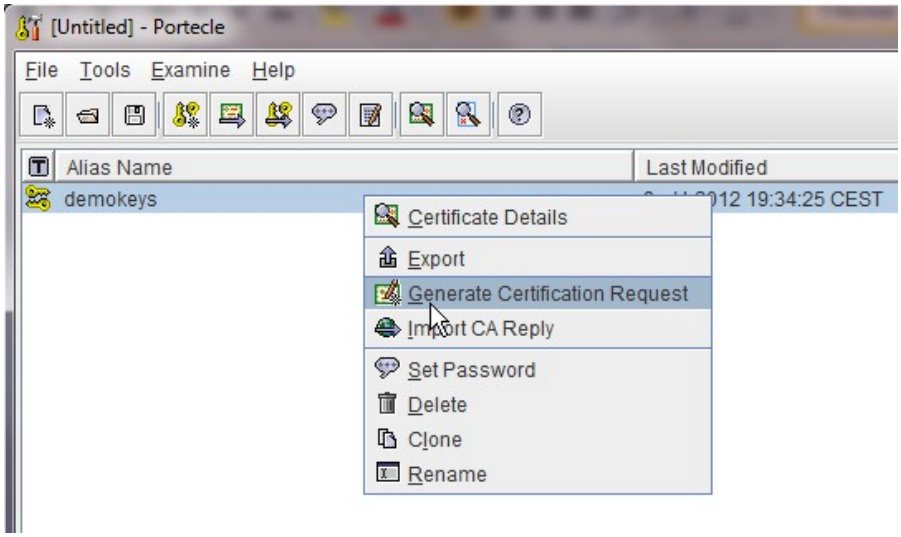


Keytool is a powerful tool, but not always easy or intuitive to use. That's why I strongly advise you to use **Portecle**, a user-friendly GUI application for creating, managing and examining keystores, keys, certificates, certificate requests, and more.

It is available on http://portecle.sourceforge.net/

You can create a new (empty) keystore with `File > New Keystore`. Then, with `Tools > Generate Key Pair`, you can generate both public and private keys.

Using your new certificate, you can `Generate a certification request` (a ***.csr** file) that you will need to send to a C.A. to have it signed.



Note that the *.**csr** file is a text file containing base 64-encoded data (you can open it with notepad). It contains only your identity and your public key, not your private key obviously. Once you have the answer from the C.A., you can import it with `Import CA Reply`.

In addition to your own keystore, you can also consult the Java **truststore** (a keystore that contains only the public keys and the identities of the C.A.). Do `File > Open CA Certs Keystore`, and enter the password (by default: `changeit`) – you'll retrieve there "DigitCert Inc".

Portecle is a powerful tool with many options (like examining files) I invite you to discover.

10.4. OpenSSL

The most powerful tool (but not the easiest to use) is OpenSSL, a console utility. Written for Unix, a version for Windows exists as well (cf. http://www.openssl.org/)

Here you start by generating a private key (a *.**pem** file)

```
openssl genrsa -out privkey.pem
```

Then you can generate a certification request (a *.**csr** file)

```
openssl req -new -key privkey.pem -out cert.csr
```

With **OpenSSL**, operations are more atomic (compare it with **KeyTool** where you generate at the same time your identity and the public/private keys). You can also easily convert the file from one format to another one (often binary vs. Base-64 text).

10.5. Using Tomcat with certificates

To install and configure **SSL** support (https + certificates) on Tomcat, you need to modify the file server.xml (in the `conf` folder), for instance:

```
<Connector  port="8443"
            protocol="org.apache.coyote.http11.Http11Protocol"
            SSLEnabled="true"
            maxThreads="150"
            scheme="https"
            secure="true"
            clientAuth="false"
            sslProtocol="TLS"
            keystoreFile="${user.home}/.keystore"
            keystorePass="changeit"
/>
```

All the information is available on http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html

10.6. File Types

Basically, there is nothing difficult with the digital certificates, but the great number of files and format makes the whole thing confusing.

The following table will help you:

Extensions	Usage
*.pem	Base64-encoded certificate, enclosed between

	---BEGIN CERTIFICATE--- and ---END CERTIFICATE---
*.cer, *.crt, *.der	Certificates binary-encoded (DER format) or Base64-encoded.
*.p12	File format commonly used to store **private keys** with accompanying **public key certificate**, protected with a **password**-based symmetric key.
*.p10, *.csr	Format of messages sent to a Certification Authority (CA) to request certification of a public key. **CSR** = Certificate Signing Request.

For other formats, see e.g. http://en.wikipedia.org/wiki/PKCS