DOSSIER DE SECURISATION DES WEB SERVICES

ANNEXE 3 : ETUDE DES TUNNELS SSL

ABREVIATIONS

Abréviation	Signification
SOAP	Simple Object Access Protocol
HTTP	Hyper Text Transfer Protocol
FTP	File Transfer Protocol
LDAP	Lightweight Directory Access Protocol
SMTP	Simple Mail Transfer Protocol
SSL	Secure Socket Layer
TLS	Transport Layer Security
PKI	Public Key Infrastructure
MAC	Message authentification code
RSA	Rivest Shamir Adleman
RC4	Rivest Cipher 4
MD5	Message Digest 5
SHA-1	Secure Hash Algorithm
DSS	Digital Signature Services
OSI	Open Systems Interconnection
VPN	Virtual Private Network
IPSec	Internet Protocol Security
(T)DES	(Triple) Data Encryption Standard
AES	Advanced Encryption Standard
IDEA	International Data Encryption Algorithm
JSSE	Java Secure Socket Extension
JDK	Java Development Kit

REFERENCES

Abréviation	Signification
[DOSSIERSECWS]	Dossier_Securisation_Web_Services_v1r0.pdf
[ANASEC]	Annexe1_Analyse_Sécurité_Préalable_v1r0.pdf
[FILTRAGE]	Annexe2_Etude_ Securistation_WS_FiltrageIP_v1r0.pdf
[SSL]	Annexe3_Etude_ Securistation_WS_TunnelSSL_v1r0.pdf
[SYNAPSE]	Annexe4_Etude_ Securistation_WS_Synapse_v1r0.pdf
[SPRING]	Annexe5_Etude_ Securistation_WS_SPRING-Security_v1r0.pdf
[PROTOTYPE]	Annexe6_Etude_ Securistation_WS_Prototype_v1r0.pdf

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

TABLE DES LISTINGS

1.AVANT PROPOS

Ce document constitue une annexe du « Dossier de sécurisation des Web Services » publié par l'AMUE. Il a pour objectif de présenter la mise en œuvre d'un tunnel SSL dans un objectif de sécurisation des Web Services. Par conséquent, afin d'appréhender correctement le contexte et le contenu de ce document, il est conseillé de lire au préalable le dossier de sécurisation des Web Services [DOSSIERSECWS].

Ce document concernant la mise en place d'un tunnel SSL s'adresse à un public connaissant déjà ce domaine proche du monde des réseaux et du système.

2.IMPLEMENTATION DE SSL POUR SECURISER LES WEB SERVICES

2.1. PRINCIPES

En se connectant sur Internet, des liens généralement sans protection sont établis entre votre ordinateur et d'autres machines. Pour minimiser les risques de sécurité informatique liés à ces échanges de données, il est possible de sécuriser les connexions, lors de l'implémentation des programmes qui les utilisent. Ce principe général s'applique aux Web Services en sécurisant le flux entre le consommateur et le fournisseur de service. Les principes de confidentialité et d'intégrité des messages sont ainsi pris en compte. Nous allons donc voir comment mettre en place une connexion sécurisée entre un client et un serveur à l'aide de TLS (SSL v.3) en Java.

2.1.1.1. SSL

Le protocole Secure Socket Layer a été développé par Netscape en 1994 puis a été repris par l' IETF en 2001. Il porte la dénomination TLS pour Transport Layer Security depuis la version 3 du SSL. Ce protocole permet des échanges de données sécurisées grâce à des algorithmes tels que RSA, et a pour objectif d'assurer l'authentification, la confidentialité et l'intégrité des données échangées mais aussi l'interopérabilité. SSL est composé d'un générateur de clés, de fonctions de hachage et d'algorithme de chiffrement (RC4, MD5, SHA-1, DSS), de protocole de gestion de session (handshake protocol) et de certificats X509.

Si on devait situer SSL sur la couche OSI, il prendrait place entre la couche transport et la couche application. De ce fait il peut encapsuler aussi bien du HTTP, du FTP, du LDAP que du SMTP. . . Cette indépendance vis à vis du protocole d'échange a largement contribué à sa diffusion et a permis la « *SSLisation* » d'un grand nombre de protocoles qui se voient tous attribuer un numéro de port spécifique par l'IANA.

De nos jours SSL est la solution privilégiée pour la sécurisation de flux et a même tendance à supplanter IPSec dans les VPN pour sa plus grande facilité de mise œuvre.

SSL assure 3 propriétés de sécurité :

- **Authentification** :
  - des parties grâce aux PKI et certificats X509 entre autres
  - des données grâce aux MACs (fonctions de hachage qui s'assurent que le message vient de l'expéditeur par l'utilisation d'une clé secrète)
- **Confidentialité** des données assurée par l'utilisation d'algorithmes à clés symétriques tel que (T)DES, AES, IDEA, RC4
- **Intégrité** des données grâce aux MACs.

SSL est subdivisé en 2 phases, aux rôles distincts :

- **Handshake** : cette phase assure la gestion de l'authentification ainsi que la négociation des paramètres de sécurité pour l'échange de données. C'est lors de celle-ci que le matériel cryptographique de la connexion KM (Key Material) est généré. Il se compose pour chaque sens de communication (Client -> Serveur, Serveur -> Client) de :
  - Une clé de session pour le chiffrement

- Un vecteur d'initialisation
- Une clé de session pour la génération des MAC

- La phase **Record** : ce protocole s'occupe du chiffrement, de la compression sans perte (optionnelle), de l'encapsulation et la fragmentation (facultative) des données. Il assure aussi les opérations inverses.

SSL fonctionne de manière simpliste comme ceci : Le serveur et le client s'échange des données pour utiliser les algorithmes que le client peut supporter. Ensuite, c'est l'échange de clés publiques, le serveur envoie un certificat au client qui contient sa clé publique, le client génère une clé publique de session chiffrée avec le certificat du serveur. Une communication chiffrée par une clé de session identique sur le client et le serveur peut alors se dérouler durant toute la session.

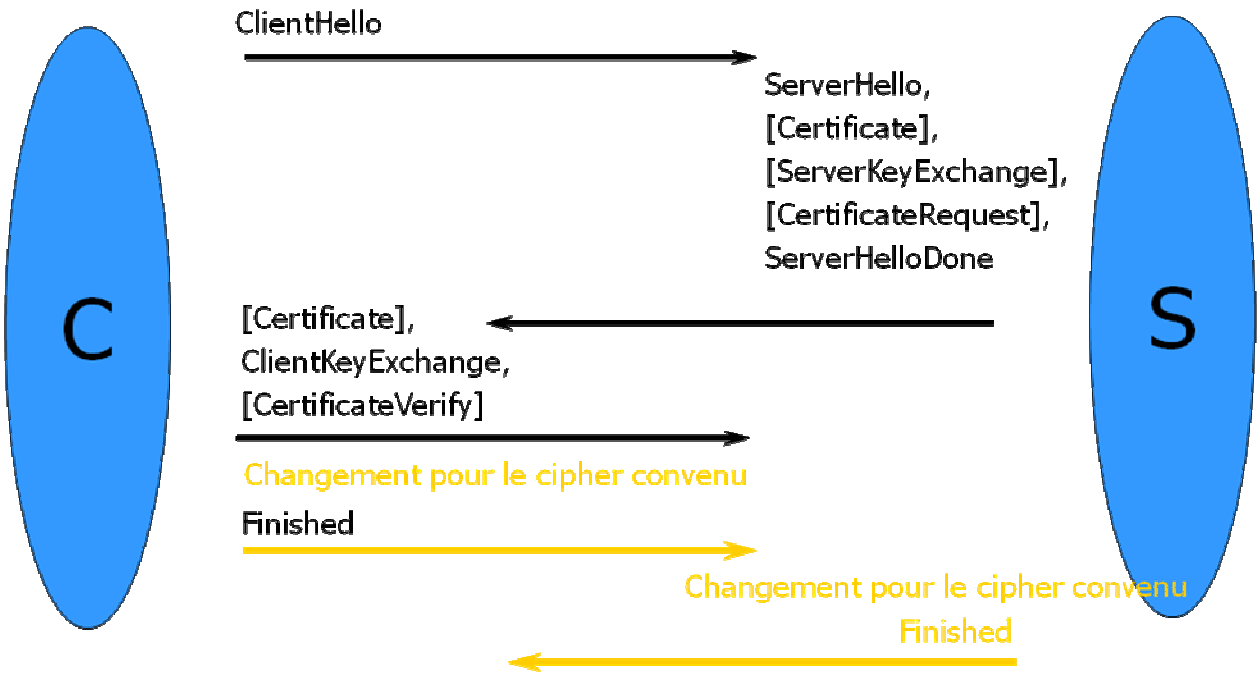

Figure 1 : Echange de clefs SSL

2.1.1.2. JSSE

Dans le monde Java, l'implémentation des protocoles SSL et TLS est JSSE (Java Secure Socket Extension). JSSE étend les API de sécurité et réseau en fournissant une classe de sockets spécialisée SSL, des gestionnaires de clés, des systèmes de fabriques de sockets mais aussi le support pour les mécanismes de négociation, d'authentification et de session. JSSE implémente RSA jusqu'à 2048 bits. JSSE était autrefois une extension mais est maintenant inclus dans le JDK 1.4.

2.2. APPLICATION

La mise en œuvre d'un tunnel SSL se réalise en 3 phases. La première est la création du ou des certificats qui permettront de chiffrer les messages et de garantir la confidentialité puis la configuration du serveur et enfin la configuration du client.

2.2.1. CREATION DES CERTIFICATS

La génération des certificats peut être réalisée avec l'outil standard de java Keytool, soit en ligne de commande, soit avec une interface graphique avec KeyTool IUI. Pour plus d'information sur Keytool (http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html). Il peut aussi être réalisé avec d'autres outils comme openssl.

Pour une authentification mutuelle, c'est-à-dire chacun présente son certificat à l'autre, il faut deux certificats. Pour une authentification simple de coté serveur il n'en faut qu'un.

Dans le cas ou des certificats sont fournis par une PKI externe, cette phase n'est pas à réaliser

Afin de générer un certificat Serveur de type RSA il faut taper :

```
keytool -genkey -alias Server -keyalg RSA -keystore server_keystore -dname "cn=Server" -keypass
password -storepass password
```

Puis ceci si vous avez besoin d'un certificat client pour réaliser une authentification mutuelle

```
keytool -genkey -alias Client -keyalg RSA -keystore client_keystore -dname "cn=Client" -keypass
password -storepass password
```

Maintenant que les certificats sont créés, vous pouvez les afficher :

```
keytool -list -keystore client_keystore
```

```
keytool -list -keystore server_keystore
```

Il faut maintenant les signer :

Le certificat client :

```
keytool -selfcert -alias Client -keystore client_keystore -keypass password -storepass password
```

Et serveur :

```
keytool -selfcert -alias Server -keystore server_keystore -keypass password -storepass password
```

Une fois les certificats signés, il faut exporter les clés publiques au format binaire dans un fichier :

La clé publique du client :

```
keytool -export -keystore client_keystore -alias Client -storepass password -file client_cert
```

Et la clé publique du serveur:

```
keytool -export -keystore server_keystore -alias Server -storepass password -file server_cert
```

Vous pouvez visualiser le contenu de ces fichiers:

```
keytool -printcert -file client_cert
keytool -printcert -file server_cert
```

Importation des clés publiques dans le magasin de clé de l'autre partie

```
keytool -import -alias Client -file client_cert -keystore server_keystore -storepass password
keytool -import -alias Server -file server_cert -keystore client_keystore -storepass password
```

Puis affichage du contenu des magasins de clés:

```
keytool -list -keystore server_keystore
keytool -list -keystore client_keystore
```

2.2.2. IMPLEMENTATION COTE CLIENT

Faire dialoguer un consommateur de web service avec un fournisseur à travers le protocole sécurisé HTTPS, nécessite un certain nombre de paramétrage du Client

Le paramétrage au niveau du client est assez simple. Ce dernier doit faire appel à certaines classes de l'API Java *JSSE* Cette API permet de manipuler en Java des sockets sécurisés répondant aux spécifications SSL. Pour plus d'information :

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

Après avoir créée un keystore, nous allons en utiliser les informations pour faire appel au web service de façon sécurisé.

```
String[][] props = {
//Chemin pour atteindre le certificat trustore (certification)
 { "javax.net.ssl.trustStore", "\Cheminverstruststore\server_cert",},
//Chemin pour atteindre le keystore
 { "javax.net.ssl.keyStore","\Cheminverskeystore\server_keystore ", },
//Mot de passe du keystore
 { "javax.net.ssl.keyStorePassword", "password", },
//Type du keystore
 { "javax.net.ssl.keyStoreType", "JKS", },
};

//Parcours des propriétés définies ci-dessus pour les intégrer dans les propriétés du système (System)
for (int i = 0; i < props.length; i++)
  System.getProperties().setProperty(props[i][0], props[i][1]);
```

Listing 1 : Configuration JSSE

Le client peut alors effectuer un appel classique au service exposé à travers HTTPS.

2.2.3. IMPLEMENTATION COTE SERVEUR

Le paramétrage de SSL au niveau du serveur est simple. Nous allons étudier 2 cas qui seront assez courant :

- Un frontal apache
- Un frontal WSO2 ESB

2.2.3.1. Frontal Apache

Afin de faire fonctionner SSL avec apache, il convient de vérifier que le module SSL est installé et chargé par Apache2

```
cd etc/apache2/mods-available
a2enmod ssl
/etc/init.d/apache2 reload
```

Le protocole SSL a besoin d'émettre sur un port spécifique pour pouvoir fonctionner, celui qui est adopté en général est le port 443. Pour cela on ajoute cette directive de configuration dans le fichier `/etc/apache2/ports.conf` :

```
Listen 443
```

Il faut ensuite configurer apache pour aller chercher les certificats générés précédemment :

(Nous utilisons la balise <code> pour représenter un virtualhost, un directory, etc. afin de ne pas être spécifique à une architecture)

```
<code>
        SSLEngine on
        SSLCertificateFile /etc/apache2/ssl/server_cert
        SSLCertificateKeyFile /etc/apache2/ssl/server_keystore
</code>
```

Listing 2 : Configuration SSL Apache

Il faut alors relancer apache et le serveur est configuré.

Dans le cas de l'utilisation d'un autre frontal web (Ex : Tomcat), la documentation sur le web est très fournie en exemple de configuration (ex : http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html)

2.2.3.2. Frontal ESB

Le WSO2 ESB nécessite une configuration particulière pour qu'il échange avec le client, en utilisant le protocole de transport sécurisé HTTPS et en passant par un service Proxy (chargé de réacheminer le message vers le web service réel). Le service Proxy étant exposé via HTTPS, il faut alors éditer le fichier *axis2.xml* du moteur Axis2 embarqué et déployé par le WSO2 ESB. Ce fichier se trouve dans le répertoire : <wso2>/*webapp**WEB-INF**classes**conf*\ et modifier la section « HTTPS Transport Sender » afin de lui préciser les différents certificats à utiliser.

```
<!--Précision de la classe du sender https-->
  <transportSender name="https" class="org.apache.synapse.transport.nhttp.HttpCoreNIOSSLSender">
    <parameter name="non-blocking" locked="false">true</parameter>
      <!-- Paramètres du keystore-->
    <parameter name="keystore" locked="false">
      <KeyStore>
        <!--Précision du nom du keystore utilisé. Il se trouve par défaut dans le répertoire
<wso2>\webapp\WEB-INF\classes\conf-->
        <Location>identity.jks</Location>
        <!--Précision du type de fichier du keystore, ici JKS : Java Key Store-->
        <Type>JKS</Type>
        <!--Mot de passe du keystore-->
        <Password>password</Password>
        <!--Mot de passe de la clé privée-->
        <KeyPassword>password</KeyPassword>
      </KeyStore>
    </parameter>
      <!--Paramètres du truststore (il s'agit du certificat partagé entre le client et le serveur) -->
    <parameter name="truststore" locked="false">
      <TrustStore>
        <!--Précision du nom du truststore utilisé. Il se trouve par défaut dans le répertoire
<wso2>\webapp\WEB-INF\classes\conf-->
        <Location>trust.jks</Location>
        <Type>JKS</Type>
        <Password>password</Password>
      </TrustStore>
    </parameter>
  </transportSender>
```

Listing 3 : Configuration SSL WSO2

Cette configuration permet au WSO2 ESB d'être capable d'échanger avec un consommateur par le protocole HTTPS par un port particulier. La figure suivante expose le WSDL du service Proxy UserProxy (et non le WSDL réel du web service UserService) où le proxy est exposé via le protocole HTTPS à travers le port *8243*.

```
- <wsdl:service name="UserProxy">
  - <wsdl:port name="UserProxyHttpSoap11Endpoint" binding="impl:UserProxySoap11Binding">
      <soap:address location="http://LCPXP-521:8280/soap/UserProxy.UserProxyHttpSoap11Endpoint"/>
    </wsdl:port>
  - <wsdl:port name="UserProxyHttpsSoap11Endpoint" binding="impl:UserProxySoap11Binding">
      <soap:address location="https://LCPXP-521:8243/soap/UserProxy.UserProxyHttpsSoap11Endpoint"/>
    </wsdl:port>
  - <wsdl:port name="UserProxyHttpsSoap12Endpoint" binding="impl:UserProxySoap12Binding">
      <soap12:address location="https://LCPXP-521:8243/soap/UserProxy.UserProxyHttpsSoap12Endpoint"/>
    </wsdl:port>
  - <wsdl:port name="UserProxyHttpSoap12Endpoint" binding="impl:UserProxySoap12Binding">
      <soap12:address location="http://LCPXP-521:8280/soap/UserProxy.UserProxyHttpSoap12Endpoint"/>
    </wsdl:port>
  - <wsdl:port name="UserProxyHttpEndpoint" binding="impl:UserProxyHttpBinding">
      <http:address location="http://LCPXP-521:8280/soap/UserProxy.UserProxyHttpEndpoint"/>
    </wsdl:port>
  - <wsdl:port name="UserProxyHttpsEndpoint" binding="impl:UserProxyHttpBinding">
      <http:address location="https://LCPXP-521:8243/soap/UserProxy.UserProxyHttpsEndpoint"/>
    </wsdl:port>
</wsdl:service>
```

Figure 2 : Exemple de configuration WSO2