A. Introduction à la sécurité du serveur et des applications

La sécurité informatique est un très vaste sujet qu'il est, aujourd'hui, impossible de ne pas traiter dans un ouvrage comme celui-ci. L'objectif de ce chapitre est de présenter les différents moyens utilisables pour sécuriser un serveur Tomcat mais également les applications qu'il héberge, et garantir un accès fiable à ses services.

B. Authentification, autorisation et cryptage : le modèle de sécurité Java EE

L'authentification est le procédé qui permet de déterminer et de valider l'identité d'un client d'une application. Les technologies Java EE, et plus particulièrement la spécification Servlet, proposent un mécanisme pour gérer cette authentification, et ce, de manière standard entre les serveurs d'applications et les conteneurs Web, grâce à l'API JAAS.

Cependant, il est tout à fait possible que les concepteurs et développeurs d'une application implémentent leur propre mécanisme pour l'authentification, cependant en cas de modification du registre contenant les données d'authentification, l'application doit s'adapter, JAAS permet de s'affranchir de cette contrainte : c'est le serveur d'application qui fait l'interface avec le registre utilisateur.

Authentification

Un conteneur Web comme Tomcat possédant un moteur HTTP peut tout à fait utiliser les mécanismes d'authentification habituellement mis en œuvre par les serveurs Web : ce sont les mécanismes d'authentification HTTP ou schémas d'authentification HTTP. Le principe est le suivant : lorsqu'un client tente d'accéder à une ressource protégée d'un site ou d'une application Web, le serveur renvoie le code d'état HTTP **401** au navigateur, indiquant à celui-ci que la ressource demandée est protégée et que son accès est soumis à authentification. En réaction à ce code **401**, le navigateur affiche une boîte de dialogue de saisie d'informations d'identification au client : si l'authentification aboutit, la ressource est envoyée dans la réponse, sinon c'est le code d'état **403 (Forbidden)** associé à une page d'erreur qui est envoyée.

Pour implémenter ce mécanisme il faut utiliser un des trois schémas d'authentification HTTP :

- L'authentification de base (**BASIC**)
- L'authentification codée (**DIGEST**)
- L'authentification par certificat client (**CLIENT-CERT**)

L'authentification de base (**Basic Auth**), est, comme son nom l'indique, le schéma d'authentification le plus simple. Lorsque le client valide les informations d'identification saisies dans la boîte de dialogue que lui présente le navigateur, ces informations sont transmises simplement en étant encodées avec l'algorithme Base64. Cet algorithme est connu et il est très facile de récupérer les données en clair. Une fois que le client est authentifié, le navigateur conserve les informations d'authentification en cache et il n'y pas d'autre moyen que de fermer le navigateur pour déconnecter le client.

L'authentification codée (**Digest Auth**) offre les mêmes fonctionnalités que l'authentification de base, mais les informations d'authentification sont cryptées grâce à un mécanisme appelé hachage (en anglais, digest). Le principe est que l'élément qui a été crypté par hachage n'est pas décryptable : le hachage est irréversible.

Le processus d'authentification codée se déroule de la manière suivante :

- Le serveur envoie la demande d'authentification au navigateur client, avec cette demande, il transmet une chaîne de caractères qui sera utilisée par le processus de hachage.
- Le navigateur ajoute cette chaîne au mot de passe du client et fait le cryptage avec un algorithme de hachage, tel que **SHA** ou **MD5**.
- Le résultat du hachage est envoyé au serveur.
- Le serveur récupère le nom d'utilisateur et le mot de passe en clair à partir du registre d'authentification. Le serveur utilise ensuite la chaîne de hachage transmise au client pour faire le hachage.
- L'authentification aboutit uniquement si les deux valeurs de hachage, celle transmise par le client, et celle générée par le serveur sont strictement identiques.

À noter que ce mécanisme ne réalise le cryptage que pour l'authentification, les données qui transitent ensuite ne sont pas cryptées.

Enfin, l'authentification par certificat client (**Client-Cert Auth**) utilise les certificats HTTPS pour garantir au client l'identité d'un serveur si le certificat est signé par un organisme digne de confiance appelé autorité de certification (des sociétés telles que VeriSign par exemple). Le serveur envoie sa clé publique au client par un

moyen ou par un autre, le client installe le certificat dans le navigateur, ensuite, les requêtes de ce client sont cryptées grâce à la clé publique du serveur, et sont décryptables uniquement par le serveur grâce à sa clé privée. Les données étant transmises en utilisant le protocole HTTPS, c'est le schéma d'authentification le plus sécurisé.

Malheureusement, aujourd'hui, tous ces mécanismes d'authentification ne sont pas supportés par tous les navigateurs Web. Dans le cas de la mise en œuvre de la sécurité dans un contexte Intranet, l'infrastructure matérielle et logicielle est maîtrisée : les postes de travail utilisateur peuvent être mis à jour pour supporter un schéma d'authentification plus qu'un autre or c'est difficilement envisageable dans un contexte Internet…

Aussi, dans la majorité des cas d'utilisation actuels, le schéma utilisé est l'authentification de base, mais les données ne sont pas véhiculées en HTTP mais en HTTPS pour crypter les flux échangés.

Un autre mécanisme d'authentification existe dans les spécifications servlet, il s'agit de l'authentification par formulaire (**Form-based Auth**). Dans ce cas, le navigateur n'intervient pas pour fournir un moyen à l'utilisateur de s'identifier, mais un formulaire HTML est utilisé par le serveur, il est présenté au client à partir du moment où celui-ci tente d'accéder à des données protégées.

Lorsque le formulaire est rempli et validé, les données d'authentification de l'utilisateur sont transmises à une servlet système du serveur d'applications, qui transmet ces données au serveur pour l'authentification dans le registre utilisateur. Ce mécanisme est très utilisé car il possède de nombreux avantages. D'abord le fait d'utiliser un formulaire HTML permet de personnaliser l'interface d'authentification, les données sont également facilement transmissibles en HTTPS, de ce fait, la sécurité repose à la fois sur une authentification par nom d'utilisateur et mot de passe, et sur le cryptage des données transmises. Un autre avantage est qu'il utilise les sessions HTTP, et qu'il est, du coup, très facile de déconnecter un client.

Autorisation

Le modèle de sécurité des applications Java EE utilise la notion d'utilisateur et de rôle pour gérer les autorisations d'accès. Le serveur d'applications qui gère le mécanisme, est lié au registre utilisateur qui contient les comptes utilisateurs, les rôles, ainsi que les associations entre ces comptes d'utilisateurs et les rôles auxquels ces utilisateurs peuvent prétendre.

Une application peut déclarer un ensemble de ressources protégées et uniquement accessibles aux utilisateurs disposant d'un rôle particulier. Un utilisateur obtient un rôle pendant son authentification : une fois son identité vérifiée, le serveur d'application ou le conteneur Web demande au registre utilisateur la liste des rôles pour cet utilisateur, ces rôles sont présentés pour l'accès à la ressource, et s'il y a correspondance, la ressource concernée est autorisée à l'utilisateur.

L'application déclare un ou plusieurs rôles pour restreindre les accès, l'association entre les rôles déclarés dans l'application, et les utilisateurs stockés dans le registre d'authentification, se fait en général au moment du déploiement de l'application. L'administrateur a également la possibilité de revenir sur cette association ultérieurement.

Cette séparation permet d'utiliser des registres d'authentification différents simplement par configuration du serveur d'applications, sans impact sur le code des applications.

Cryptage

Le cryptage des données entre un client et un serveur Web se fait en utilisant une variante du protocole HTTP qui transite dans un canal sécurisé grâce à un protocole réseau qui crypte les données.

SSL (*Secure Socket Layer*) est un protocole qui permet de transmettre des données de manière sécurisée entre un client et un serveur. Développé par la société Netscape Inc., SSL a été très rapidement adopté en tant que protocole standard de l'Internet.

Le protocole SSL repose sur deux mécanismes de sécurité, le cryptage à clé publique et le cryptage à clé symétrique.

Le cryptage à clé publique fait intervenir une paire de clé, la clé publique, qui est librement diffusée et disponible, et la clé privée qui est soigneusement conservée par son propriétaire. Tout ce qui est crypté avec la clé privée n'est décryptable qu'avec la clé publique, et inversement, tout ce qui est crypté avec la clé publique n'est décryptable qu'avec la clé privée.

Le cryptage à clé symétrique utilise le même mécanisme pour le cryptage et le décryptage des données. Cependant, il ajoute également un procédé permettant à un client de récupérer la clé publique du serveur en toute sécurité.

En effet, un pirate pourrait tout à fait se faire passer pour le serveur auquel le client souhaite se connecter, envoyer la clé publique au client, et récupérer des informations sensible sur ce dernier. Le mécanisme de cryptage par clé publique ne permet pas au client de s'assurer qu'il dialogue bien avec le bon serveur.

Dans le cryptage symétrique, les étapes d'échange de la clé publique entre le serveur et le client sont les suivantes :

- Le serveur commence par envoyer un certificat au client qui fait la demande de communication, ce certificat contient la clé publique du serveur, l'émetteur du certificat, et la durée de validité du certificat.
- Le client doit ensuite accepter le certificat en fonction de son authenticité. Un certificat peut être considéré authentique à partir du moment où il est délivré par un organisme digne de confiance appelé autorité de certification (**Certificate Authorities - CA**), comme les sociétés **VeriSign** ou encore **Thawte**. Si le certificat n'est pas valide, le client en est averti, libre à lui de continuer ou d'arrêter le dialogue, à ses risques et périls.
- À ce stade, les échanges entre le client et le serveur sont maintenant sécurisés par le mécanisme de cryptage à clé publique. La configuration du serveur peut, ou non, demander une authentification au client : les données d'authentification qui transitent sont cryptées.

Le protocole **HTTPS** (*HTTP over SSL*) est une implémentation de HTTP sur le protocole **SSL**. Une autre implémentation de protocole sécurisé a également vu le jour, il s'agit de **TLS** (*Transport Layer Security*). C'est la version standardisée de **SSL** par l'**IETF** (*Internet Engineering Task Force*), l'organisation la plus importante en matière de standardisation des protocoles de l'Internet. **TLS** s'appuie sur la version 3.0 de **SSL**. En général, les serveurs d'applications JEE peuvent utiliser **SSL** ou **TLS** pour sécuriser les flux HTTP.

Les technologies Java utilisent une implémentation des protocoles **SSL** et **TLS** dans la bibliothèque **JSSE** (*Java Secure Socket Extension*). Cette extension de la plate-forme Java fait partie intégrante du J2SE depuis la version 1.4, pour les versions précédentes, il faut télécharger cette extension sur le site de Sun Microsystems (**http://java.sun.com/products/jsse/**), et installer les fichiers de la bibliothèque dans le répertoire **JAVA_HOME/jre/lib/ext**.

1. La sécurité des applications Web Java EE

Pour permettre à une application Web Java EE d'utiliser les mécanismes décrits dans la partie précédente, il est nécessaire de la configurer en tant que telle. L'objectif est de définir quelles sont les ressources à protéger, à quels utilisateurs seront-elles rendues exclusivement accessibles, et comment les clients vont-ils s'identifier sur l'application.

Cette configuration se fait dans le descripteur de déploiement de l'application Web, le fichier **web.xml**.

L'élément de configuration XML `<security-constraint>` permet à la fois de déclarer les ressources à protéger avec l'élément enfant `<web-resource-collection>`, et les rôles qui auront accès à ces ressources, avec l'élément enfant `<auth-constraint>`. Ensuite, le mécanisme d'authentification est déclaré avec un autre élément XML : `<login-config>`. La section se termine avec la déclaration de tous les rôles de sécurité utilisés dans l'application, avec l'élément `<security-role>`.

L'exemple qui suit est un fragment de descripteur de déploiement (fichier **web.xml**) qui montre une configuration complète de sécurité Java EE.

```
<security-constraint>
   <display-name>Contraintes de sécurité</display-name>
   <web-resource-collection>
      <web-resource-name>Accès restreint</web-resource-name>
      <description></description>
      <url-pattern>/admin/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>PUT</http-method>
      <http-method>HEAD</http-method>
      <http-method>TRACE</http-method>
      <http-method>POST</http-method>
      <http-method>DELETE</http-method>
      <http-method>OPTIONS</http-method>
   </web-resource-collection>
   <auth-constraint>
      <description></description>
      <role-name>administrateurs</role-name>
   </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>FORM</auth-method>
```