LinuxFocus article number 285
http://linuxfocus.org

Accéder à PostgreSQL par JDBC via un tunnel SSL Java

par Chianglin Ng
<chglin(at)singnet.com.sg>

L´auteur:

Je vis à Singapour, un pays moderne et cosmopolite situé en Asie du Sud-Est. J'utilise Linux depuis environ deux ans. Ma première distribution a été une RedHat 6.2. Aujourd'hui j'utilise une RedHat 8.0 chez moi et parfois une Debian Woody.

Traduit en Français par:
Georges Tarbouriech
<gt(at)linuxfocus.org>

Résumé:

Cet article montre comment établir un accès JDBC à PostgreSQL sur RedHat 8.0 et comment créer un tunnel SSL grâce aux extensions "Secured Socket" Java de Sun, afin de permettre un accès distant sécurisé à la base de données.

Introduction

Pendant l'apprentissage de Postgres et de JDBC, j'ai découvert le problème de l'accès distant sécurisé à une base de données par l'intermédiaire de JDBC.
Les connexions JDBC ne sont pas cryptées et un "sniffer" de réseau peut facilement intercepter des données sensibles. Plusieurs méthodes existent pour se protéger. Le manuel de postgres informe de la possibilité de le compiler avec le support de SSL ou d'utiliser un tunnel SSH.

A la place de ces méthodes, je préfèrerais me servir de Java. Le JDK 1.4.1 de Sun contient les extensions Java Secured Socket (JSSE) qui peuvent être utilisées pour créer des connexions SSL. Le JDK propose également un outil de génération de clés privée/publique (keytool), de certificats et de stockage de ces clés. A partir de là, il devient relativement facile de construire un couple de proxies basés sur Java afin de sécuriser le transfert des données sur le réseau.

Installer PostgreSQL pour JDBC sur RedHat 8.0

Les instructions proposées ici sont destinées à RedHat 8.0 mais les principes de base restent applicables à d'autres distributions. Vous devez installer PostgreSQL et les pilotes JDBC correspondants. Sur RedHat 8.0, vous pouvez utiliser les rpm ou l'outil graphique de gestion de paquetages. Vous devez aussi télécharger et installer le JDK 1.4.1 de Sun. Le JDK de Sun comporte quelques restrictions de cryptage dues à la règlementation Américaine sur l'exportation. Pour bénéficier d'un cryptage illimité, vous pouvez télécharger les fichiers JCE (Java Cryptographic Extensions). Visitez le site Java de Sun pour plus ample information.

J'ai installé le JDK 1.4.1 dans /opt et défini la variable d'environnement JAVA_HOME pour la faire correspondre à mon répertoire JDK. J'ai aussi modifié la variable PATH pour y ajouter le chemin vers le répertoire contenant les exécutables JDK. Voici les lignes ajoutées à mon fichier .bash_profile file.

```
JAVA_HOME = /opt/j2sdk1.4.1_01
PATH = /opt/j2sdk1.4.1_01/bin:$PATH
export JAVA_HOME PATH
```

Les fichiers limitant le cryptage livrés avec le JDK de Sun ont été remplacés par les illimités de JCE. Pour permettre à Java de trouver les pilotes JDBC pour postgres, j'ai copié les pilotes postgres-jdbc dans mon répertoire d'extensions Java (/opt/j2sdk1.4.1_01/jre/lib/ext). Dans RedHat 8.0, les pilotes postgres-jdbc se trouvent dans /usr/share/pgsql.

S'il s'agit de votre première installation de PostgreSQL, vous devrez créer une nouvelle base de données et un nouveau compte d'utilisateur PostgreSQL. Logez-vous en tant que root par su et démarrez le service postgres. Modifiez ensuite le compte administrateur postgres par défaut.

```
su root
password:******
[root#localhost]#/etc/init.d/postgresql start
[root#localhost]# Starting postgresql service: [ OK ]
[root#localhost]# su postgres
[bash]$
```

Créez un nouveau compte postgres et une base de données.

```
[bash]$:createuser
Enter name of user to add: chianglin
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) y
CREATE USER
[bash]$createdb chianglin
CREATE DATABASE
```

J'ai créé un nouveau compte d'administrateur postgres qui correspond à mon compte d'utilisateur Linux ainsi qu'une nouvelle base de données du même nom. Par défaut, lorsque vous lancez l'outil psql, il se connecte à une base correspondant au compte de l'utilisateur Linux actuel. Voir le manuel de postgres pour plus de détails sur l'administration des comptes et des bases. Pour attribuer un mot de passe au

compte nouvellement créé, vous pouvez exécuter psql et lancer la commande ALTER USER. Logez-vous sous votre nom d'utilisateur et démarrez psql. Tapez ce qui suit

ALTER USER chianglin WITH PASSWORD 'test1234' ;

Pour autoriser les connexions TCP/IP, vous devez éditer le fichier postgresql.conf et définir l'option tcpip_socket option à true (vrai). Sur RedHat 8.0, ce fichier se trouve dans /var/lib/pgsql/data. Passez root et tapez ce qui suit

tcpip_socket=true

La dernière étape consiste à éditer le fichier pg_hba.conf. Il définit les hôtes autorisés à se connecter à la base de données postgres. J'ai ajouté un seul hôte contenant l'adresse loopback de ma machine et utilisant une authentification par mot de passe. Vous devez être root pour modifier ce fichier.

host sameuser 127.0.0.1 255.255.255.255 password

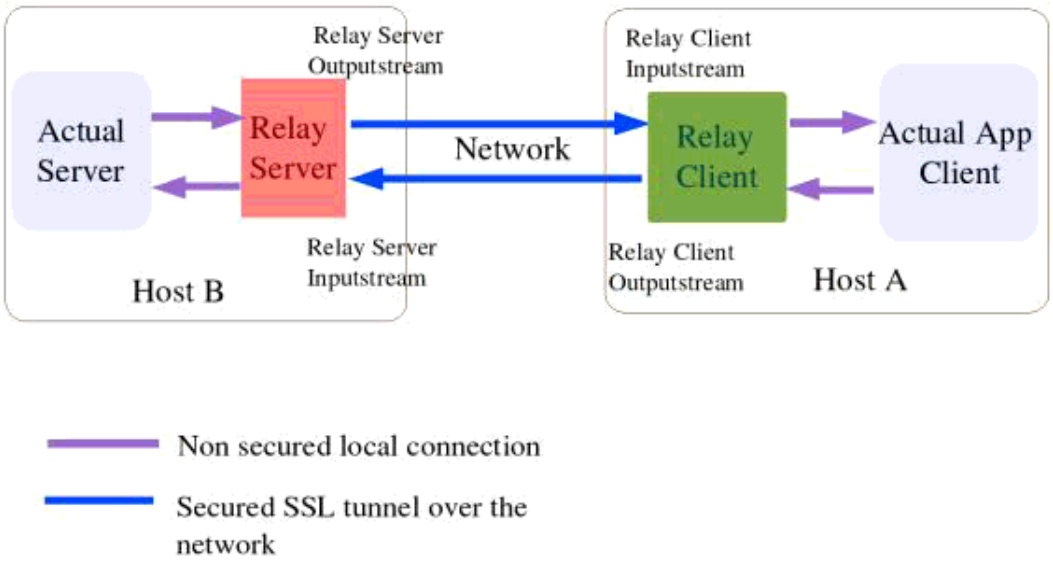
Redémarrez postgres pour activer les changements.

Création du tunnel SSL Java

A la suite de l'étape qui précède, postgres est prêt à accepter les connexions locales JDBC non sécurisées. Pour accéder à postgres de manière sécurisée et à distance, il est nécessaire de recourir à une forme de relais.

Le graphique suivant montre comment doit fonctionner ce type de relais.

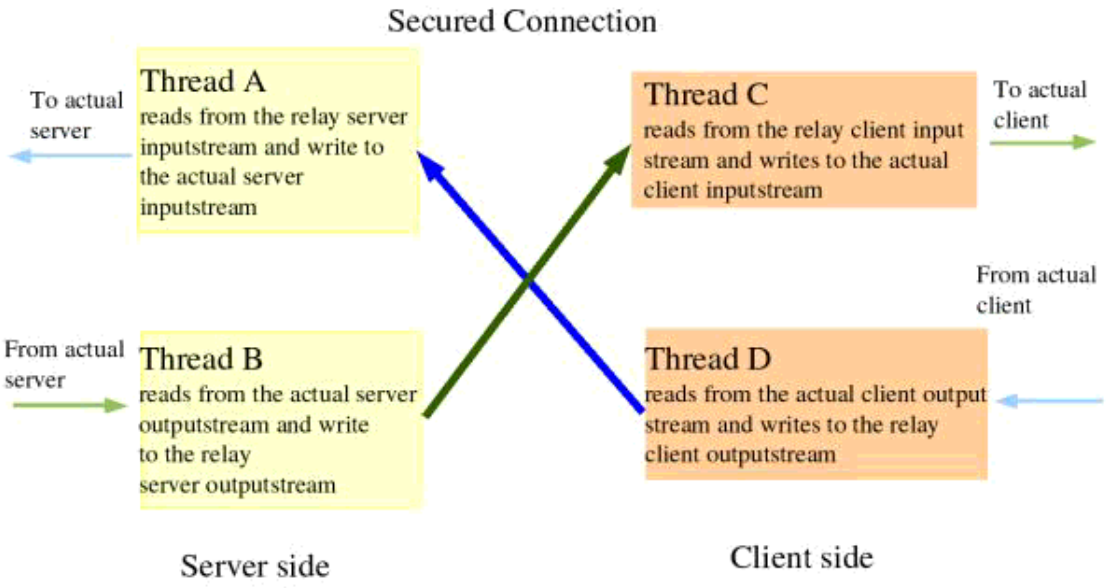
Figure 1. Client / Server Secured Relaying



L’application JDBC doit se connecter au client proxy qui relaiera les données par une connexion SSL vers le serveur proxy distant. Le serveur proxy acheminera les paquets vers postgres et enverra les réponses par la connexion SSL au client proxy qui les relaiera vers l’application JDBC. Le processus sera transparent pour l’application JDBC.

Sur le graphique on se rend compte que côté serveur, il est nécessaire de récupérer les données du flux sécurisé entrant et de les envoyer vers le flux sortant local connecté au serveur. L’inverse est aussi vrai, puisque vous devez récupérer les données du flux entrant local connecté au serveur et les router vers le flux sortant sécurisé. Cela s’applique aussi au client. Pour la mise en oeuvre, on peut utiliser des "threads". La figure suivante le montre.

Fig 2. Threads needed



Création des "magasins de clés", des clés et des certificats

Une connexion SSL réclame habituellement une authentification du serveur. L'authentification du client est optionnelle. Dans notre cas, je préfère avoir une authentification des deux. Cela signifie que je vais devoir créer des certificats et des clés pour le client et pour le serveur. Pour ce faire je me sers de l'outil (keytool) fourni par le JDK. J'aurai deux "magasins" de clés sur le client et sur le serveur. Le premier est utilisé pour stocker la clé privée de l'hôte et le second pour stocker les certificats auxquels l'hôte fait confiance.

Ce qui suit montre la création d'un "magasin" de clés, d'une clé privée et d'un certificat public auto-signé pour le serveur.

keytool -genkey -alias serverprivate -keystore servestore -keyalg rsa -keysize 2048

Enter keystore password: storepass1
What is your first and last name?
[Unknown]: ServerMachine
What is the name of your organizational unit?
[Unknown]: ServerOrg
What is the name of your organization?
[Unknown]: ServerOrg
What is the name of your City or Locality?
[Unknown]: Singapore
What is the name of your State or Province?
[Unknown]: Singapore

What is the two-letter country code for this unit?
[Unknown]: SG
Is CN=ServerMachine, OU=ServerOrg, O=ServerOrg, L=Singapore, ST=Singapore, C= [no]: yes
Enter key password for <serverprivate>
(RETURN if same as keystore password): prikeypass0 </serverprivate>

Remarquez que les mots de passe sont réclamés deux fois. La première fois c'est pour le "magasin" et la deuxième fois c'est pour la clé privée. Ensuite, il faut exporter vers un fichier le certificat public du serveur qui sera utilisé par le client pour authentifier le serveur en question.

keytool -export -alias serverprivate -keystore -rfc servestore -file server.cer

Ce qui précède exporte le certificat public auto-signé du serveur vers le fichier server.cer. Côté client, il faut importer ce fichier dans un "magasin" contenant tous les certificats publics dans lesquels le client a confiance.

keytool -import -alias trustservercert -file server.cer -keystore clienttruststore

La commande ci-dessus importe le certificat public du serveur dans un "magasin" nommé clientruststore. Si le "magasin" n'existe pas, il sera créé et un mot de passe sera réclamé.

A ce stade, votre système est capable d'utiliser une connexion SSL pour l'authentification du serveur. Puisque je veux également authentifier le client, je dois créer une clé privée/publique pour le client dans un nouveau "magasin", exporter le certificat public du client et l'importer dans un nouveau "magasin" côté serveur.

A la fin de cette étape, il devrait y avoir deux "magasins" de clés sur le serveur, l'un contenant sa clé privée et l'autre les certificats dans lesquels il a confiance. C'est la même chose pour le client.

Afin d'exécuter l'exemple que je propose plus loin, il est essentiel de donner le même mot de passe à chaque "magasin" créé sur une machine donnée. Autrement dit, les deux "magasins" de clés du serveur doivent être protégés par le même mot de passe. Il en est de même pour les deux "magasins" du client.

Vous pouvez consulter la documentation de Sun pour en savoir plus sur l'outil keytool.

Mise en oeuvre des classes

Mes classes utilisent les extensions Java Secured Socket de Sun. Le guide de référence des JSSE de Sun est disponible à http://java.sun.com/j2se/1.4.1/docs/guide/security/jsse/JSSERefGuide.html. Pour la connexion SSL, il vous faut une instance de l'objet SSLContext fourni par JSSE. Initialisez ce SSLContext avec les paramètres qui vous conviennent et vous obtenez une classe Secured SocketFactory. La 'socketfactory' est utilisée pour créer les sockets SSL.

Dans mon exemple, il y aura une classe proxy pour le client et le serveur afin de construire le tunnel SSL. Comme ils utiliseront tous les deux une connexion SSL, ils hériteront d'une classe de base SSLConnection. Cette classe aura la charge de définir le SSLContext initial utilisé par les proxies du client et du serveur. Enfin, il nous faut une autre classe pour gérer les "threads" servant de relais. Quatre classes en tout.

Ce qui suit montre un extrait du code de la classe SSLConnection

Extrait de la classe SSLConnection

```
/* initKeyStore method to load the keystores which contain the private key and the trusted certificates */

public void initKeyStores(String key , String trust , char[] storepass)
{
      // mykey holding my own certificate and private key, mytrust holding all the certificates that I trust
  try {
      //get instances of the Sun JKS keystore
     mykey = KeyStore.getInstance("JKS" , "SUN");
     mytrust = KeyStore.getInstance("JKS", "SUN");

    //load the keystores
   mykey.load(new FileInputStream(key)  ,storepass);
   mytrust.load(new FileInputStream(trust) ,storepass );
    }
 catch(Exception e) {
    System.err.println(e.getMessage());
    System.exit(1);
    }
}

/* initSSLContext method to obtain a  SSLContext and initialize it with the SSL protocol and data from
the keystores */
public void initSSLContext(char[] storepass , char[] keypass) {
    try{
    //get a SSLContext from Sun JSSE
    ctx = SSLContext.getInstance("TLSv1" , "SunJSSE") ;
    //initializes the keystores
    initKeyStores(key , trust , storepass) ;

    //Create the key and trust manager factories for handing the cerficates
    //in the key and trust stores
    TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509" ,
    "SunJSSE");
    tmf.init(mytrust);

    KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509" ,
    "SunJSSE");
    kmf.init(mykey , keypass);

    //initialize the SSLContext with the data from the keystores
    ctx.init(kmf.getKeyManagers() , tmf.getTrustManagers() ,null) ;
    }
    catch(Exception e) {
    System.err.println(e.getMessage());
```

```
   System.exit(1);
   }

}
```

La méthode initSSLContext créé un SSLContext à partir des JSSE de Sun. Pendant la création, vous pouvez préciser le protocole SSL à utiliser. Dans notre cas, j'ai choisi TLS (Transport Layer Security) version 1. Dès qu'une instance de SSLContext est obtenue, ele est initialisée avec les données des "magasins" de clés.

L'extrait de code suivant vient de la classe SSLRelayServer qui sera exécutée sur la même machine que la base de données postgres. Elle relaiera les données du client venant de la connexion SSL vers postgres et vice versa.

La classe SSLRelayServer

```
/* initSSLServerSocket method will get the SSLContext via its super class SSLConnection. It will then
create a  SSLServerSocketFactory object that will be used to create a SSLServerSocket. */

public void initSSLServerSocket(int localport) {
     try{
         //get the ssl socket factory
         SSLServerSocketFactory ssf = (getMySSLContext()).getServerSocketFactory();

          //create the ssl socket
         ss = ssf.createServerSocket(localport);
         ((SSLServerSocket)ss).setNeedClientAuth(true);
     }
  catch(Exception e) {
     System.err.println(e.getMessage());
     System.exit(1);
   }
 }

// begin listening on SSLServerSocket and wait for incoming client connections
public void startListen(int localport , int destport) {

   System.out.println("SSLRelay server started at " + (new Date()) + "  " +
                "listening on port " + localport + "  " +  "relaying to port " + destport );

 while(true) {
     try {
       SSLSocket incoming = (SSLSocket) ss.accept();
       incoming.setSoTimeout(10*60*1000); // set 10 minutes time out
       System.out.println((new Date() ) + " connection from " + incoming );
       createHandlers(incoming, destport); // create 2 new threads to handle the incoming connection
      }
```

```
    catch(IOException e ) {
       System.err.println(e);
       }
    }
}
```

La classe RelayApp, le proxy client, est semblable à SSLRelayServer. Elle hérite de SSLConnection et utilise 2 "threads" pour effectuer le relais. La différence vient de ce qu'elle créé un SSLSocket pour se connecter à l'hôte distant au lieu d'un SSLServerSocket pour écouter les connexions entrantes. La dernière classe requise concerne le "thread" effectuant le véritable relais. Elle se contente de lire les données d'un flux entrant et de les écrire dans un flux sortant.

Le code exemple complet des quatre classes est disponible ici (example285-0.1.tar.gz).

Lancement des proxies et test

Sur le client, vous devez avoir les fichiers SSLConnection.java, RelayIntoOut.java et RelayApp.java. Sur le serveur, il vous faut SSLRelayServer.java, RelayIntoOut.java et SSLConnection.java. Regroupez-les dans un répertoire. Pour compiler le proxy client, tapez la commande suivante.

javac RelayApp.java

Pour compiler le proxy serveur, tapez

javac SSLRelayServer.java

Sur le serveur exécutant postgres, vous pouvez lancer SSLRelayServer avec 6 arguments de la ligne de commande. Ce sont

1. Le chemin complet vers le "magasin" contenant la clé privée du serveur précédemment créée par keytool
2. Le chemin complet vers le "magasin" du serveur contenant le certificat du client de confiance
3. Le mot de passe de vos "magasins" de clés
4. Le mot de passe de la clé privée de votre serveur
5. Le port écouté par le serveur relais
6. Le port vers lequel rediriger les données (le port du serveur réel, dans notre cas postgresql, qui est par défaut 5432)

java SSLRelayServer servestore trustclientcert storepass1 prikeypass0 2001 5432

Quand le proxy serveur est lancé, vous pouvez démarrer le proxy client. Le proxy client réclame 7 arguments, l'argument supplémentaire concernant le nom d'hôte ou l'adresse IP du serveur sur lequel se connecter. Les arguments sont les suivants

1. Le chemin complet vers le "magasin" contenant la clé privée du client
2. Le chemin complet vers le "magasin" du client contenant le certificat du serveur de confiance
3. Le mot de passe de vos "magasins" de clés

4. Le mot de passe de la clé privée de votre client
5. Le nom d'hôte ou l'adresse IP du serveur
6. Le numéro de port du serveur relais de destination (2001 dans l'exemple ci-dessus)
7. Le numéro de port de l'application à relayer, dans ce cas postgresql, il devrait donc être défini à 5432

java RelayApp clientstore trustservercert clistorepass1 cliprikeypass0 localhost 2001 5432

Une fois le tunnel SSL établi, vous pouvez démarrer l'application JDBC et vous connecter à postgres de la manière habituelle. Le processus de relais sera totalement transparent pour l'application JDBC. Cet article est déjà trop long et je ne proposerai donc pas d'exemples d'application JDBC. Le manuel de postgres et le tutoriel de Sun contiennent de nombreux exemples concernant JDBC.

Si vous voulez tout faire sur une seule machine pour tester, vous pouvez aussi. Il existe deux méthodes différentes : soit vous demandez à la base postgres d'écouter un autre port, soit vous redirigez le numéro de port utilisé par RelayApp vers un autre port. J'utiliserai cette dernière méthode pour illustrer un simple test. Quittez d'abord RelayApp; vous devez tuer la tâche en pressant [Ctrl] c. Vous faites de même pour arrêter le proxy SSLRelayServer proxy.

Relancez RelayApp par la commande suivante. La seule différence concerne le dernier numéro de port qui est maintenant 2002.

java RelayApp clientstore trustservercert clistorepass1 cliprikeypass0 localhost 2001 2002

La meilleure application de test n'est autre que psql. Nous allons relayer tout le trafic de psql vers postgres par notre tunnel. Tapez la commande suivante pour démarrer psql à des fins de test.

psql -h localhost -p 2002

La commande indique à psql de se connecter au "localhost" sur le port 2002 écouté par RelayApp. Après avoir tapé le mot de passe de postgres, vous pouvez commencer à exécuter des commandes SQL et à tester la connexion SSL qui sert de relais.

Une remarque sur la sécurité

Ce n'est pas une bonne idée de taper les mots de passe comme argument de la ligne de commande si vous partagez une machine. Ceci parce que si quelqu'un exécute la commande ps -auxww il pourra voir la chaîne entière du lancement de processus, y compris le mot de passe. Il est préférable de stocker les mots de passe de manière cryptée dans un autre fichier afin que votre application java aille les y lire. Une alternative consiste à utiliser Java Swing pour créer une boîte de dialogue réclamant le mot de passe.

Conclusion

Les JSSE de Sun sont simples d'emploi pour la création d'un tunnel SSL utilisable par postgres. De fait, n'importe quelle autre application nécessitant une connexion sécurisée peut certainement utiliser ce tunnel SSL. Il existe de très nombreuses façons d'ajouter le cryptage à vos connexions; lancez simplement votre éditeur Linux favori et commencez à coder. Amusez-vous bien !

Liens utiles

- Code source de cet article
- Documentation PostgreSQL
- Spécifications des JSSE de Sun
- Spécifications du JCA de Sun
- Tutoriel sur la sécurité avec Java

Site Web maintenu par l´équipe d´édition LinuxFocus	Translation information:
© Chianglin Ng	en --> -- : Chianglin Ng <chglin(at)singnet.com.sg>
"some rights reserved" see linuxfocus.org/license/	en --> fr: Georges Tarbouriech <gt(at)linuxfocus.org>
http://www.LinuxFocus.org	

2005-01-14, generated by lfparser_pdf version 2.51