Benutzung von PostgreSQL über JDBC mittels Java–SSL–Tunnel

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

About the author:

Ich lebe in Singapore, ein modernes, vielrassiges Land in Südostasien. Linux benutze ich seit ungefähr zwei Jahren. Mit Red Hat 6.2 fing ich an, jetzt benutze ich Red Hat 8.0 zu Hause. Gelegentlich arbeite ich auch mit Debian Woody.

Abstract:

Dieser Artikel zeigt, wie man JDBC–Zugang für PostgreSQL in Red Hat 8.0 einrichtet und wie man mittels Sun's Java Secured Socket Extensions einen SSL–Tunnel herstellt, der sicheren Zugang zu einer entfernten Postgres–Datenbank ermöglicht.

Einleitung

Als ich mich mit Postgres und JDBC beschäftigte, stiess ich auf das Problem einen sicheren Zugang zu einer entfernten Datenbasis mittels JDBC zu erhalten.
JDBC–Verbindungen sind nicht verschlüsselt und ein Netzwerk –Eindringling kann ziemlich einfach Zugang zu vertraulichen Daten bekommen. Es gibt verschiedene Möglichkeiten, das zu verhindern. Das Postgres–Handbuch erwähnt, dass man Postgres mit SSL–Unterstützung kompilieren kann oder SSH–Tunneling benutzt.

Anstelle dieser Methoden möchte ich Java selbst benutzen. Sun's Java JDK 1.4.1 enthält die Java Secured Socket Extensions (JSSE), damit ist es möglich SSL–Verbindungen herzustellen. JDK liefert auch ein Verschlüsselungswerkzeug zur Herstellung von öffentlichen und privaten Schlüsseln, digitalen Zertifikaten und Schlüsselspeichern. Infolgedessen ist es verhältnismässig einfach, ein Paar auf Java basierende Proxys (Relay) zu bauen, über welche Netzwerkdaten sicher übertragen werden können.

PostgreSQL für JDBC in Red Hat 8.0 einrichten

Die nachfolgenden Anweisungen gelten für Red Hat 8.0, das allgemeine Konzept ist auch für andere Distributionen anwendbar. Du mußt jedoch PostgreSQL und die dazugehörigen JDBC−Treiber installieren. Bei Red Hat 8.0 kannst du dafür den **rpm**−Befehl oder das Management −Tool benutzen. Du mußt auch Sun's JDK 1.4.1 herunterladen und installieren. Dank der US−Exportbestimmungen kommt Sun's JDK mit eingeschränkten Verschlüsselungsmöglichkeiten. Für unbegrenzte Verschlüsselung kannst du die Dateien mit den JCE (Java Cryptographic Extensions) herunterladen. Auf Sun's Java Website findest du weitere Einzelheiten.

Ich habe JDK 1.4.1 in */opt* installiert und die JAVA_HOME −Environment−Variable zeigt auf mein JDK−Verzeichnis. Mein PATH habe ich ebenfalls aktualisiert, dort befinden sich jetzt die ausführbaren Dateien von JDK. Hier die Zeilen, die ich meiner *.bash_profile* − Datei hinzugefügt habe.

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

Die begrenzten Verschlüsselungsdateien von Sun JDK habe ich durch die uneingeschränkten JCE−Dateien ersetzt. Damit Java die JDBC−Treiber für Postgres finden kann, kopierte ich diese in mein Java−Unterverzeichnis (*/opt/j2sdk1.4.1_01/jre/lib/ext*). In Red Hat 8.0 sind die Postgres−JDC−Treiber in */usr/share/pgsql* zu finden.

Falls das deine erste Postgres−Installation ist, musst du eine neue Datenbasis und ein neues Postgresql − Benutzerkonto einrichten. Als erstes mit **su** nach **root** und den Postgres−Service starten, dann zum Default − Postgres − Administratorkonto.

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

Einrichten eines neuen Postgres−Konto und einer Datenbasis.

```
[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
```

Mein neu eingerichtes Postgres−Administrator−Konto korrespondiert mit meinem Linux−Benutzerkonto und einer Datenbasis gleichen Namens. Das psql − Werkzeug verbindet sich in der Grundeinstellung mit der Datenbasis, die mit dem gegenwärtigen Linux−Benutzerkonto korrespondiert. Weitere Einzelheiten zur Verwaltung von Konten und Datenbanken sind im Postgres−Handbuch zu finden. Zur Eingabe deines Passworts für das neue Konto, starte psql und gib den Befehl *ALTER USER* ein. Melde dich in deinem normalen Benutzerkonto an und starte psql. Gib folgendes ein

```
ALTER USER chianglin WITH PASSWORD 'test1234' ;
```

Um die tcpip −Verbindungen zu ermöglichen, bearbeite *postgresql.conf* und stelle die *tcpip_socket*−Option auf "true". In Redhat 8.0 ist diese Datei in */var/lib/pgsql/data* zu finden. Wechsle in **root** und mach die folgende Einstellung

tcpip_socket=true

Der letzte Schritt ist die Bearbeitung der Datei *pg_hba.conf*. Sie bestimmt, welche Hosts sich mit der Postgres−Datenbank verbinden können. Mit einer einzigen Eingabe für den Host habe ich die Loop−Adresse angepasst, einschliesslich Authentifizierung mittels Passwort. Die Anpassung der Datei muss von **root** aus erfolgen.

host sameuser 127.0.0.1 255.255.255.255 password

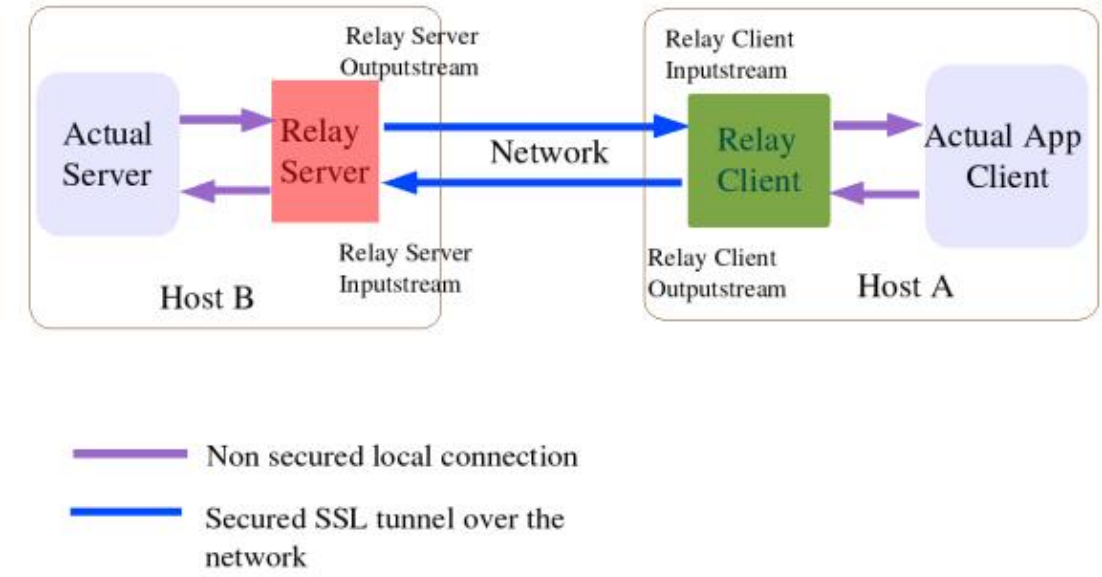
Mit dem Neustart von Postgres werden die neuen Einstellungen aktiv.

Entwickeln des Java – SSL – Tunnels

Nach den vorangegangenen Schritten ist Postgres bereit, unsichere JDBC−Verbindungen zu akzeptieren. Der Fernzugriff auf Postgres muss über ein Proxy (Relay) erfolgen.

Die folgende Darstellung zeigt, wie die Proxy−Übergabe (Relay) funktionieren sollte.

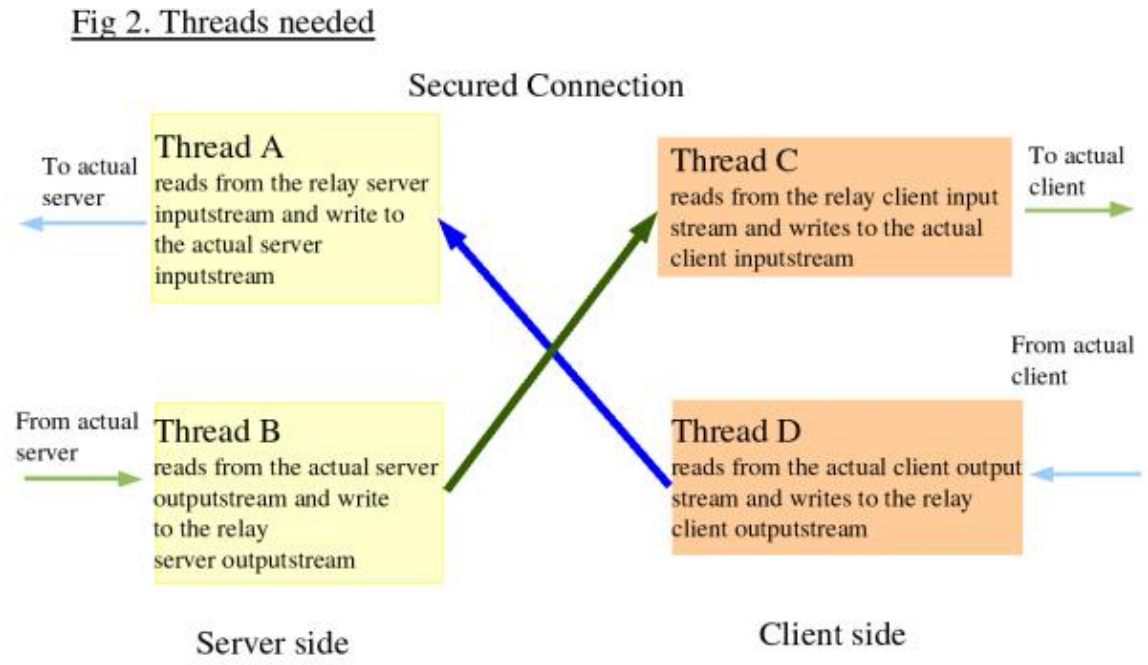
Figure 1. Client / Server Secured Relaying



Die JDPC−Anwendung verbindet sich mit dem Client−Proxy, welcher dann alle Daten über eine SSL−Verbindung auf den entfernten Proxy−Server übermittelt. Der Proxy−Server schickt einfach alle Pakete zu Postgres und die Antwort erfolgt über die SSL−Verbindung zurück an den Client−Proxy, der sie zur

JDBC−Anwendung überträgt. Der gesamte Vorgang geschieht transparent für die JDBC−Anwendung.

Wie in der Abbildung angedeutet, besteht für den Server die Notwendigkeit, die eintreffenden Daten in einem sicheren Stream zu empfangen und an den lokalen Ausgabe−Stream zu übergeben, der mit dem aktuellen Server verbunden ist. Umgekehrt trifft das auch zu: du benötigst die Daten vom lokal eintreffenden Stream des aktuellen Servers und leitest diese zum sicher übertragenden Stream. Das gleiche Schema gilt für den Client − es kann mittels Threads durchgeführt werden. Die folgende Abbildung macht das deutlich.



Anlegen von Schlüsselspeichern, Schlüsseln und Zertifikaten

Eine SSL−Verbindung erfordert Server−Authentifizierung. Client−Authentifizierung ist wahlweise. In unserem Fall ziehe ich beides vor, d.h. ich muss Zertifikate und Schlüssel für den Server und den Client anlegen. Dafür benutze ich das Keytool im Java JDK. Auf dem Client und dem Server lege ich zwei Schlüsselspeicher an. Der erste Speicher wird für den privaten Schlüssel des Hosts und der zweite für die Zertifikate, denen der Host vertraut, angelegt.

Nachfolgend wird gezeigt, wie man einen Schlüsselspeicher, einen privaten Schlüssel und ein öffentliches, selbstbestätigendes Zertifikat für den Server anlegt.

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>

Hier ist zu bemerken, dass das Passwort zweimal benötigt wird. Erstens für den Schlüsselspeicher und zweitens für den privaten Schlüssel. Danach exportiere das öffentliche Zertifikat des Servers, welches der Client zur Authentifizierung des Servers benutzen wird, in eine Datei.

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

Hiermit exportieren wir des Servers selbstbestätigendes, öffentliches Zertifikat in die Datei *server.cer*. Auf Seiten des Client importieren wir diese Datei in einen Schlüsselspeicher, der alle öffentlichen Zertifikate enthält, denen der Client vertraut.

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

Mit diesem Befehl wird das öffentliche Zertifikat des Servers in einen Schlüsselspeicher namens clienttruststore importiert. Falls dieser noch nicht besteht, wird er erzeugt und du wirst aufgefordert, ein Passwort für den Speicher einzugeben.

Jetzt ist dein System bereit, eine SSL−Verbindung mittels Server−Authentifizierung herzustellen.
Da ich auch den Client authentifizieren will, muss ich auch einen privaten und einen öffentlichen Schlüssel für den Client in einem neuen Client−Schlüsselspeicher einrichten, danach das öffentliche Zertifikat des Client in einen neuen Server−Schlüsselspeicher auf dem Server exportieren.

Am Ende dieses Prozesses sollten auf dem Server und dem Client je zwei Schlüsselspeicher zu finden sein, einer enthält den privaten Schlüssel, der andere das vertraute Zertifikat.

Um das nachfolgende Code−Beispiel ausführen zu können, ist es notwendig, das gleiche Passwort für jeden der Schlüsselspeicher auf der entsprechenden Maschine einzurichten. Das bedeutet, die zwei Schlüsselspeicher des Servers sollten das gleiche Passwort haben, das gleiche gilt für die beiden Schlüsselspeicher des Client.

Weitere Informationen zum *keytool* sind in Sun's Dokumentation zu finden.

Einführung der Klassen

Meine Klassen werden von Sun's Java Secured Socket Extensions Gebrauch machen. Die JSSE − Referenz finden wir hier. Für eine SSL−Verbindung benötigst du die Instanz eines SSL−Kontext−Objekts, das von JSSE geliefert wird. Initialisiere diesen SSL−Kontext mit den gewünschten Einstellungen und du erhältst daraus eine Secured SocketFactory − Klasse. Mit der SocketFactory kann man die SSL− Sockets erzeugen.

Für meine Anwendung wird eine Client− und eine Server− Proxy−Klasse zur Zusammenstellung des SSL−Tunnel benötigt. Da sie beide eine SSL−Verbindung benutzen werden, werden sie eine SSL−Verbindungsklasse erben. Diese Klasse wird dafür zuständig sein, den ursprünglichen SSL−Kontext

einzurichten, der vom Client – sowie vom Server–Proxy benutzt werden wird. Zusätzlich benötigen wir noch eine weitere Klasse, um die übertragenden Threads aufzubauen. Insgesamt also vier Klassen.
Hier ein Codeabschnitt aus der SSL–Verbindungsklasse

Codeabschnitt aus der SSL–Verbindungsklasse

```
/* 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);
   }
```

```
}
```

Die initSSL−Kontext−Methode erzeugt einen SSL−Kontext mittels Sun's JSSE. Dabei kannst du das gewünschte SSL−Protokoll angeben, ich habe TLS Version 1 (Transport Layer Security) ausgewählt. Sobald wir eine Instanz des SSL−Kontext haben, wird diese mit den Daten der Schlüsselspeicher initialisiert.

Der folgende Codeabschnitt stammt von der SSLRelayServer−Klasse, die auf der gleichen Maschine laufen wird wie die Postgres−Datenbank. Damit werden alle Clientdaten über die SSL−Verbindung nach Postgres übertragen und umgekehrt.

SSLRelayServer−Klasse

```
/* 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);
       }
   }
}
```

Die RelayApp−Klasse, d.h. der Client−Proxy, ähnelt dem SSLRelay−Server. Er erbt von der SSL−Verbindung zwei Threads und benutzt diese, um die Übertragung durchzuführen. Der Unterschied liegt darin, dass er einen SSL−Sockel aufbaut, um mit dem entfernten Host zu verbinden, anstatt einen SSLServer−Sockel, der auf Verbindungsaufforderungen wartet. Die letzte benötige Klasse ist der Thread, der die eigentliche Übertragung durchführt. Er liest einfach den Eingabestrom und schreibt diesen in einen Ausgabestrom.

Das vollständige Codebeispiel für die vier Klassen ist hier zu finden (example285−0.1.tar.gz).

Die Proxys starten und testen

Auf dem Client benötigst du die Dateien SSLConnection.java, RelayIntoOut.java und RelayApp.java. Der Server benötigt SSLRelayServer.java, RelayIntoOut.java und SSLConnection.java. Speichere alle in einem Verzeichnis. Führe folgenden Befehl aus, um den Client−Proxy zu kompilieren.

javac RelayApp.java

Um den Server zu kompilieren, gib folgenden Befehl ein

javac SSLRelayServer.java

Mit Postgres auf deinem Server installiert, kannst du den SSLRelayServer mit sechs Kommandozeilen−Argumenten starten. Hier sind sie

1. Der vollständige Pfad zum Schlüsselspeicher mit dem privaten Schlüssel, den du vorher mit Keytool erzeugt hast.
2. Vollständiger Pfad zum Schlüsselspeicher des Servers, der die vertraulichen Client−Zertifikate enthält.
3. Passwort für die Schlüsselspeicher (keystore)
4. Passwort für deinen privaten Schlüssel zum Server
5. Der Port, auf dem der Relay−Server wartet
6. Der Port, auf den die Daten geschickt werden (in diesem Fall *postgresql* mit der Grundeinstellung 5432)

java SSLRelayServer servestore trustclientcert storepass1 prikeypass0 2001 5432

Sobald der Server−proxy läuft, kannst du den Client−proxy starten. Der Client−proxy benötigt 7 Argumente, das zusätzliche ist der hostname oder die IP Adresse des Servers, zu dem du dich verbindest. Die Argumente sind:

1. Der vollständige Pfad zum Schlüsselspeicher mit dem privaten Schlüssel, den du vorher mit Keytool erzeugt hast.
2. Vollständiger Pfad zum Schlüsselspeicher des Servers, der die vertraulichen Client−Zertifikate enthält.
3. Password für den Schlüsselspeicher
4. Passwort für deinen privaten Schlüssel zum Server
5. Hostname oder IP Adresse des Servers
6. Portnummer des Ziel− relay server (in obigen Beispiel ist das 2001)
7. Portnummer der Applikation, zu der du übergibst, in diesem Fall postgresql, deshalb setzt du es auf 5432

java RelayApp clientstore trustservercert clistorepass1 cliprikeypass0 localhost 2001 5432

Sobald der SSL−Tunnel existiert, kannst du die JDBC−Anwendung starten und ganz normal Postgres öffnen. Der gesamte Übertragungsvorgang wird für die JDBC−Anwendung vollständig transparent sein.

Dieser Artikel ist bereits zu lang, ich werde daher keine weiteren Beispiele zur JDBC−Anwendung aufführen. Das Postgres−Handbuch und die Anleitung von SUN enthalten zahlreiche Beispiele zu JDBC.

Du kannst das Testen auch auf einer einzigen Maschine durchführen. Dafür bestehen zwei Möglichkeiten: entweder du änderst den Input−Port der Postgres−Datenbank oder du wechselst die Portnummer, auf die RelayApp überträgt. Ich werde den letzteren Fall anwenden, um einen einfachen Test zu demonstrieren. Zuerst schließe RelayApp mit dem kill −Befehl [strg] c. Auf die gleiche Weise kann der SSLRelayServer−Proxy geschlossen werden.

Starte wieder RelayApp mit dem folgenden Befehl, der einzige Unterschied ist die letzte Port−Nummer, sie ist jetzt 2002.

java RelayApp clientstore trustservercert clistorepass1 cliprikeypass0 localhost 2001 2002

Die beste Anwendung zum Testen ist psql selbst. Wir werden allen psql−Verkehr zu Postgres durch unseren Tunnel übertragen. Gib den folgenden Befehl ein, um psql zu für den Test zu starten

psql −h localhost −p 2002

Dieser Befehl weist psql an, mit dem Localhost am Port 2002, dem RelayApp zuhört, zu verbinden. Nach der Eingabe des Postgres−Passwort kannst du wie gewöhnlich SQL−Befehle ausführen und damit die Übertragung der SSL−Verbindung testen.

Ein Hinweis zur Sicherheit

Es ist keine gute Idee, Passwörter als Befehlszeilenargumente zu benutzen, wenn deine Maschine auch von anderen benutzt wird. Es ist möglich, mit dem Befehl *ps −auxww* den gesamten String des Prozesses, einschliesslich der Passwörter, einzusehen. Es ist besser, die Passwörter in verschlüsselter Form in einer anderen Datei zu speichern und deine Java−Anwendung diese von dort lesen zu lassen. Als Alternative besteht die Möglichkeit, mittels Java Swing ein Dialogfeld mit Eingabeaufforderung anzulegen.

Fazit

Es ist ziemlich einfach, einen SSL−Tunnel mittels Sun−JSSE zu bauen, den Postgres benutzen kann. Wahrscheinlich kann jede andere Anwendung, die eine sichere Verbindung benötigt, diese Art SSL−Tunnel benutzen. Es gibt so viele Möglichkeiten, deine Verbindungen zu verschlüsseln − starte deinen Linux−Lieblingseditor und fang an zu kodieren! Viel Spass !

Nützliche Links

- Sourcecode für diesen Artikels
- PostgreSQL Dokumentation
- Sun JSSE Spezifikationen
- Sun JCA Spezifikationen
- Java Sicherheit – Tutorial

Translation information:
en −−> −− : Chianglin Ng <chglin(at)singnet.com.sg>
en −−> de: Jürgen Pohl <sept.sapins/at/verizon.net>

2005−01−11, generated by lfparser_pdf version 2.51