

Java DB Security

Security Features in Java DB Release 10.4

Rick Hillegas
Senior Staff Engineer, Sun Microsystems

June 2008

Abstract

Java DB provides several ways to protect the correctness and privacy of your data as well as forestall accidental or malicious misuse of the database software itself. This white paper explains how to improve the database security of applications and machines which use Java DB.

Java DB

Table of Contents

Introduction.....	4
Vulnerabilities.....	4
Threats.....	5
Java DB Defenses.....	6
Defenses outside Java DB.....	6
Defenses Mapped to Threats.....	7
Safer Application Design.....	8
Appendix A: Configuring Database Encryption.....	8
Appendix B: Configuring SSL/TLS.....	9
Appendix C: Configuring Authentication.....	12
Appendix D: Configuring Coarse-grained Authorization.....	14
Appendix E: Configuring Fine-grained Authorization.....	17
Appendix F: Configuring Java Security.....	20
Appendix G: Putting It All Together.....	25
Glossary.....	29

Introduction

This white paper surveys vulnerabilities in an unsecured Java DB, threats which exploit those vulnerabilities, and defenses against those threats. This is not a complete list—no survey of security concerns can hope to be complete. However, this paper attempts to list the major vulnerabilities and threats known today. At the end of this paper, a series of appendixes explains how to configure Java DB's defenses. Please skip ahead to the appendixes if you are already familiar with Java DB's vulnerabilities and simply need a primer on how to configure Java DB security.

The author gratefully acknowledges several earlier descriptions of Java DB/Derby security:

1. Masoud Kalali's [white paper](http://today.java.net/pub/a/2007/03/20/javadb-end-to-end-security.html) describing the security features of Java DB release 10.2 (<http://today.java.net/pub/a/2007/03/20/javadb-end-to-end-security.html>).
2. Jean Andersen's description of the security features of Derby 10.1, delivered at [ApacheCon USA 2004](http://db.apache.org/derby/papers/ApacheCon.html#ApacheCon+US:+December+10-14,+2005) (<http://db.apache.org/derby/papers/ApacheCon.html#ApacheCon+US:+December+10-14,+2005>).
3. Dan Debrunner's description of the security features of Derby 10.0, delivered at [ApacheCon USA 2004](http://db.apache.org/derby/papers/ApacheCon.html#ApacheCon+US:+November+13-17,+2004) (<http://db.apache.org/derby/papers/ApacheCon.html#ApacheCon+US:+November+13-17,+2004>).

Vulnerabilities

If you do not configure Java DB security, be aware of the following vulnerabilities:

1. **Network JDBC** – Network JDBC connections expose sensitive operations to use by persons who may not have account privileges on the database machine.
2. **Cleartext Traffic** - By default, network traffic travels in cleartext.
3. **Unbounded Growth** - Tables can grow arbitrarily large.
4. **CPU Hogging** – Unbounded CPU cycles can be consumed by connection attempts, SQL queries, and user code running in the database.
5. **Superusers** - By default, all Java DB users enjoy extensive powers to read and write in all databases.
6. **Launch Privileges** - Java DB procedural code executes with the operating system privileges of the account which launched the virtual machine. This includes system-supplied procedures as well as custom, user-coded procedures.
7. **User Code** - Arbitrary user code can execute in the Java DB virtual machine via user-coded functions and procedures.
8. **Open Source** - Java DB's code itself is publicly visible as part of the Apache Derby open source project. This means

that a hacker can write subtle malware after studying the code and file formats. Note that while closed source code enjoys the advantage of “security by obscurity”, openness can confer other, countervailing security advantages. See, for instance the following [article](http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/open-source-security.html) (<http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/open-source-security.html>).

Threats

A threat is potential *damages* caused by an *attacker* using a *technique* to exploit a *vulnerability*. We have already seen examples of Java DB vulnerabilities. Examples of damages, attackers, and techniques follow.

Significant damages include:

1. **Denial-of-service** – An attacker can monopolize resources on the host machine. For instance, an attacker can launch a runaway procedure on the Java DB virtual machine, fill up the file system, or pepper the Java DB server with incessant connection requests.
2. **Theft** – An attacker can read private information stored in a Java DB database or transmitted across the network. With enough privileges and by exploiting application code visible on the classpath, an attacker can use Java DB to read private information stored elsewhere on the server machine or even on other machines inside the firewall.
3. **Corruption** – An attacker can modify or destroy information stored in a Java DB database or elsewhere inside the firewall.

Attackers include:

1. **Insiders** – These are privileged persons who enjoy access to systems inside the firewall and maybe even to restricted machine rooms. Drunken DBAs and disgruntled co-workers can cause significant damage.
2. **Outsiders** – These include politically motivated governments and guerillas, commercially motivated businesses and criminals, as well as thrill-seeking hackers.

Techniques include:

1. **SQL Injection** – This technique plagues applications which construct queries by concatenating input from clients. A clever client can put SQL into these fields. That SQL, not intended by the application, then runs inside the database.
2. **Man-in-the-middle** – In this technique, the client believes that it is talking to the server. In reality, the connection has been intercepted by another machine or sniffer which monitors traffic between the client and server. The device in the middle can examine and alter the traffic. An example of this technique is password sniffing, in which a machine in the middle intercepts the credentials handshake between client and server.
3. **Malware** – This is a general term for viruses, worms, trojans, and other intrusive/destructive code which can infect a machine.
4. **Probing** – This is the technical equivalent of jiggling door handles to see what doors are unlocked.

5. **Physical Access** – This refers to the low-tech, brute-force technique of gaining physical access to a restricted area or machine and, for instance, exploiting superuser powers which might be available from a system's console.
6. **Social Engineering** – This refers to the low-tech technique of gaining and abusing the confidence of someone who has the keys.

Java DB Defenses

In discussing Java DB defenses, the following terms are handy:

System Administrator – This is the person who configures Java DB's system-wide behavior. Typically, this is a highly privileged user responsible for allocating machine resources, managing the network, configuring security, and actually launching the VM.

Database Owner – This is the person who creates and tends the databases needed by a particular application. Of course, the Database Owner could be the System Administrator.

User – This is a person authorized to use an application. This includes end-users, technical support engineers, and developers who maintain the application.

Java DB provides the following defenses against threats. Defenses typically managed by the System Administrator are marked in **blue**. Defenses typically managed by the Database Owner are marked in **orange**.

1. **Java Security** – Using a Java SecurityManager and policy file, the System Administrator can restrict the permissions granted to user-written code. The System Administrator can also restrict the permissions granted to Java DB itself.
2. **SSL/TLS** – The System Administrator can require that SSL/TLS be used to encrypt network traffic between Java DB clients and servers, along the way raising an extra authentication hurdle.
3. **Encryption** - A Database Owner can require that the data for an application be encrypted before being stored on disk. This makes it expensive to steal and corrupt the data.
4. **Authentication** – Using usernames and passwords, a Database Owner can restrict access to an application's data.
5. **Coarse-grained Authorization** – A Database Owner can divide an application's users into three groups: those with no privileges, those with read-only privileges, and those with read-write privileges.
6. **Fine-grained SQL Authorization** – Via SQL GRANT/REVOKE commands, a Database Owner can further restrict access to fine-grained pieces of data and code.

Defenses outside Java DB

In addition to the defenses provided by Java DB, you should take advantage of defenses provided by your machine and intranet. It is important to configure these defenses to protect Java DB from attacks by both outsiders and insiders.

1. **Firewalls** – Limit network access to the machine which runs Java DB.
2. **Accounts** – Limit login access to the machine which runs Java DB. Centrally administer accounts using, for instance, an LDAP server.
3. **Physical Locks** – Limit physical access to the machine which runs Java DB.
4. **Secure Traffic** – Encrypt the traffic which flows on your internal network.
5. **File Permissions** – Restrict the file permissions granted to the account which launches Java DB.
6. **Quotas** – Limit the file space and CPU which an account can consume.

Defenses Mapped to Threats

The following table maps defenses to examples of threats which they parry. This matrix can help you decide whether you need to configure specific defenses. Consult this table if you decide NOT to configure a defense--make sure that you are still shielded from the corresponding threats. In this table, “ditto” repeats the comment in the cell above.

<i>Defense</i>	<i>Damages</i>	<i>Attacker</i>	<i>Technique</i>	<i>Vulnerability</i>
Java Security	Denial-of-service, theft, and corruption.	Insiders and outsiders.	Malware, physical access.	Network JDBC, unbounded growth, launch privileges, user code, open source.
SSL/TLS	Theft and corruption.	Insiders and outsiders.	Man-in-the middle, physical access.	Network JDBC, cleartext traffic.
Encryption	Theft and corruption.	Chiefly insiders.	Physical access.	Open source.
Authentication	Theft, corruption, denial of service.	Insiders and outsiders.	Probing.	Superusers.
Coarse-grained Authorization	ditto	ditto	ditto	ditto
Fine-grained SQL Authorization	ditto	ditto	ditto	ditto
Firewalls	Theft, corruption, denial of service.	Insiders and outsiders.	Probing.	Network JDBC.
Accounts	Theft, corruption, denial of service.	Insiders.	Man-in-the-middle, malware, physical access.	Launch privileges, user code.
Physical locks	ditto	ditto	ditto	ditto
Secure Traffic	Theft. Corruption.	Insiders.	Man-in-the-middle.	Cleartext traffic.
File Permissions	Theft, corruption, denial of service.	Insiders and outsiders.	Malware.	Launch privileges, user code, open source.

Safer Application Design

The following tips should help you write and deploy safer applications which use Java DB:

1. **Launch Account** – Create an operating system account for the System Administrator. This will be the account which launches Java DB. This account should not be the operating system's superuser.
2. **File Permissions** – Limit the file permissions of this System Administrator account to just the directories which the application should be allowed to read and write. Do not grant read or write access on these directories to any other operating system accounts.
3. **Policy File** – Write your own Java Security policy which restricts the directories which Java DB can access and the sockets on which it can accept connections.
4. **JDBC Leaks** – Do not let JDBC connections leak outside your intranet's firewall. If possible, design your application so that external clients talk to an application server, which in turn communicates with Java DB. Limit the JDBC connections to communication between the application server and Java DB.
5. **Injection** – Do not construct queries by concatenating strings which are filled in by clients. To parameterize your queries, use JDBC ? parameters in PreparedStatements.
6. **Shields Up** – By default, enable all defenses mentioned in this white paper. If you need to turn off a defense for performance reasons, then carefully consider how you will protect your application from the threats which that defense parries.

Appendix A: Configuring Database Encryption

By default, Java DB stores its data unencrypted in ordinary operating system files. An attacker who can view those files can simply type them out, exposing all sorts of data stored in string columns. Knowing Java DB's file formats, a clever attacker could even view numeric data stored in those files. Even worse, a clever attacker could change the data itself.

Fortunately, Java DB can encrypt databases. On a shared machine, that helps protect data from other users, including disgruntled or curious superusers. Encryption helps protect private financial data from thieves who physically steal your laptop.

Before encrypting a database, you need to make two choices:

1. **Boot Password** – This is the password which unlocks your encrypted data when you want to use it.
2. **Encryption Algorithm** – This is a *transformation* name as described in the Javadoc for the [javax.crypto.Cipher](#) class. Java DB encryption relies on the JCE libraries supplied with the virtual machine. For more information on those libraries, please see the [Java Cryptography Extension Reference Guide](#) (<http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html>).

Here's a little **ij** script which creates an encrypted database. Note the additional attributes on the database creation URL: *dataEncryption*, *encryptionAlgorithm*, and *bootPassword*.

```
connect
'jdbc:derby:myEncryptedDatabaseName;create=true;dataEncryption=true;encryptionAlgorithm=Blowfish/CBC/NoPadding;bootPassword=mySuperSecretBootPassword';
```

Once you have created an encrypted database, you can work in it. After you shutdown the encrypted database, you can re-connect to it by simply supplying your boot password in the connection URL, as shown in the following little **ij** script:

```
connect 'jdbc:derby:myEncryptedDatabaseName;bootPassword=mySuperSecretBootPassword';
```

Note that by booting a database with its boot password, you unlock the database for the lifetime of the virtual machine. That means that other threads can connect to the database without supplying the boot password. This situation lasts until the database is explicitly shut down or the virtual machine exits. For a single-user, shrink-wrapped application, this is generally not a problem. However, for a multi-user application, you need to keep the following in mind:

1. **Unlock** – The boot password is used to initially unlock encrypted data. Once the Database Owner has unlocked the database, other users can connect to it without supplying the boot password.
2. **Work** – For that reason, you should configure Java DB Authorization (see below) to restrict the users who may access the unlocked data.
3. **Relock** – To relock your data, simply shut down the database.

Appendix B: Configuring SSL/TLS

By default, network traffic travels in cleartext between Java DB clients and servers. Via a man-in-the-middle ploy, a clever attacker can read all of the string data shipped to and from the server. By knowing the Java DB wire protocol, a clever attacker can read numeric data too. Even worse, the man-in-the-middle can change the data in-flight.

Fortunately, Java DB can encrypt network traffic using the SSL/TLS (Secure Socket Layer/Transport Layer Security) logic supplied with the virtual machine. As a side-effect, SSL/TLS raises an extra authentication hurdle too. Before using this encryption technology, you will want to familiarize yourself with SSL/TLS concepts such as key pairs and certificates, and with the Sun JDK's *keytool* application. Useful overviews of SSL/TLS may be found at [Apache](http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html) and [Wikipedia](http://en.wikipedia.org/wiki/Secure_Sockets_Layer) (http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html and http://en.wikipedia.org/wiki/Secure_Sockets_Layer, respectively). Sun supplies *keytool* documentation for [Unix](#) and for [Windows](#) (<http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/keytool.htm> and

<http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/keytool.html>, respectively). Network encryption requires the following setup:

1. **Client certificates** – Each client must generate a client key pair and certificate. The client certificates must be loaded into the server's trust store.
2. **Server certificate** – The server must generate a server key pair and certificate. All of the clients must load the server's certificate into their respective trust stores.
3. **Server startup** – The server must be booted with system properties and a startup option which turn on SSL/TLS encryption.
4. **Client startup** – The client must be started with system properties which turn on SSL/TLS encryption. In addition, an extra attribute is added to the JDBC connection URL.

To use SSL/TLS to encrypt Java DB's network traffic, the client must have a *key store* for holding its own public/private key pair. The client must also have a *trust store* for holding the server's certificate. If the *key store* and *trust store* do not already exist, the *keytool* program will create them. Let's say the client stores its public/private key pair in `~/vault/ClientKeyStore` and let's say the client stores certificates from other systems in `~/vault/ClientTrustStore`.

Here's how you create a client key. First you must pick a password for the *key store*. Let's say this password is "secretClientPassword". Then issue the following command to create the client's public/private key pair. You will be prompted to enter the password plus various identifying information (your input is marked in **blue**):

```
keytool -genkey -alias MyClientName -keystore ~/vault/ClientKeyStore
```

```
Enter keystore password: secretClientPassword
```

```
What is your first and last name?
```

```
[Unknown]: Gertrude Stein
```

```
What is the name of your organizational unit?
```

```
[Unknown]: Proofreading Department
```

```
What is the name of your organization?
```

```
[Unknown]: Albatross Books
```

```
What is the name of your City or Locality?
```

```
[Unknown]: New York
```

```
What is the name of your State or Province?
```

```
[Unknown]: NY
```

```
What is the two-letter country code for this unit?
```

```
[Unknown]: US
```

```
Is CN=Gertrude Stein, OU=Proofreading Department, O=Albatross Books, L=New York, ST=NY, C=US correct?
```

```
[no]: yes
```

Enter key password for <MyClientName>

(RETURN if same as keystore password):

Next, create a certificate for this client:

```
keytool -export -alias MyClientName -keystore ~/vault/ClientKeyStore -rfc -file ClientCertificate -storepass secretClientPassword
```

This will create a file called ClientCertificate. Later, you will import this file into the server's *trust store*.

Now perform a similar series of steps on the server machine to create a server key pair in a *key store* guarded by the "secretServerPassword" password, and create a ServerCertificate from this key:

```
keytool -genkey -alias MyServerName -keystore ~/vault/ServerKeyStore
```

Enter keystore password: secretServerPassword

...

```
keytool -export -alias MyServerName -keystore ~/vault/ServerKeyStore -rfc -file ServerCertificate -storepass secretServerPassword
```

Now import the server certificate into the client's *trust store*:

```
keytool -import -alias favoriteServerCertificate -file ServerCertificate -keystore ~/vault/ClientTrustStore \
-storepass secretClientTrustStorePassword
```

...and import the client certificate into the server's *trust store*:

```
keytool -import -alias Client_1_Certificate -file ClientCertificate -keystore ~/vault/ServerTrustStore -storepass secretServerTrustStorePassword
```

That concludes steps (1) and (2), the setup. On to step (3). Every time that we bring up the server, we must remember to turn on network encryption. Note that we set four VM properties which declare the locations and passwords for the server's *key store* and *trust store*: *javax.net.ssl.keyStore*, *java.net.ssl.keyStorePassword*, *javax.net.ssl.trustStore*, and *javax.net.ssl.trustStorePassword*. In addition, we specify the *-ssl peerAuthentication* startup option.

```
java -Djavax.net.ssl.keyStore=/Users/me/vault/ServerKeyStore \
-Djavax.net.ssl.keyStorePassword=secretServerPassword \
-Djavax.net.ssl.trustStore=/Users/me/vault/ServerTrustStore \
-Djavax.net.ssl.trustStorePassword=secretServerTrustStorePassword \
org.apache.derby.drda.NetworkServerControl start -p 8246 -ssl peerAuthentication
```

Finally, on to step (4). We bring up a client. As with server startup, we must tell the VM the locations and passwords of the local *key store* and *trust store*. This example is a simple *ij* script. Note the extra *ssl* attribute on the connection URL. That attribute tells the client to authenticate the server's identity via a certificate and it tells the client that the network traffic must be encrypted:

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
```

```

-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.tools.ij

ij version 10.4

ij> connect 'jdbc:derby://localhost:8246/testdb;create=true;ssl=peerAuthentication';

ij> select schemaName, authorizationID from sys.sysschemas;

```

You will get errors from **ij** if you do not specify the extra VM properties and/or if you do not specify the `ssl` attribute on the connection URL. Here, for instance is the output from running **ij** without the VM properties and `ssl` attribute:

```

java org.apache.derby.tools.ij

ij version 10.4

ij> connect 'jdbc:derby://localhost:8246/testdb;create=true';

ERROR 08006: A network protocol error was encountered and the connection has been terminated: A PROTOCOL Data Stream Syntax Error
was detected. Reason: 0x3. Plaintext connection attempt to an SSL enabled server?

```

When you want to administer the server (for instance, to bring it down), you will need to specify the locations and passwords of a valid *key store* and *trust store* as well as the extra `ssl` option on the server command line:

```

java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.drda.NetworkServerControl shutdown -p 8246 -ssl peerAuthentication

```

Appendix C: Configuring Authentication

By default, Java DB runs without any credentials checking. This situation may be fine for many shrink-wrapped, embedded applications. However, it means that anyone can connect to an unsecured database and steal or corrupt the data there. Fortunately, it's easy to frustrate these attacks by requiring authentication. Java DB supports three kinds of authentication schemes:

1. **Builtin** – In this scheme, username/password pairs are declared in the system configuration file, inside individual databases, or in the VM startup command. This scheme is useful for testing an application. Its usefulness in production, however, is limited.
2. **Hand-rolled** – In this scheme, the customer provides all of the logic needed to authenticate users.
3. **LDAP** – In this scheme, the customer points Java DB at an external LDAP service. The customer manages users via

the external LDAP service and Java DB retrieves credentials from LDAP.

This appendix describes how to authenticate users with the OpenDS LDAP server. To start out with, just launch the OpenDS [QuickSetup JNLP \(Java Web Start\) installer](http://opends.org/promoted-builds/latest/install/QuickSetup.jnlp), and follow the installation steps to set up your directory server. As part of this installation, you will specify a password (<http://opends.org/promoted-builds/latest/install/QuickSetup.jnlp>). We refer to this below as `YOUR_SELECTED_PASSWORD`.

Next, load some credentials into the directory server. Download this sample file of credentials: [secArticle.LDIF](http://today.java.net/today/2007/03/22/secArticle.LDIF) (<http://today.java.net/today/2007/03/22/secArticle.LDIF>). Now load it into your directory server using the `import-ldif` tool that lives in the bin directory of your OpenDS installation. (Make sure that OpenDS is not running when you import credentials; otherwise you will receive an error message indicating that the import utility cannot acquire a lock over storage.)

```
import-ldif --backendID userRoot --ldifFile secArticle.LDIF
```

Now bring up the OpenDS server by running the `start-ds` script in the bin directory of your OpenDS installation.

Next, point Java DB at the OpenDS LDAP server by adding the following lines to your Java DB configuration file (`derby.properties`):

```
derby.connection.requireAuthentication=true
derby.authentication.server=ldap://127.0.0.1:1389
derby.authentication.provider=LDAP
derby.authentication.ldap.searchAuthPW=YOUR_SELECTED_PASSWORD
derby.authentication.ldap.searchAuthDN=cn=Directory Manager
derby.authentication.ldap.searchBase=dc=example,dc=com
derby.authentication.ldap.searchFilter=objectClass=person
```

Finally, start `ij` in the directory where you created your `derby.properties` (this ensures that embedded Java DB will come up with the authentication settings listed above). Run the following commands:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb;create=true;user=tquist;password=tquist';
```

Verify that authentication works by trying to connect again, this time with bad credentials:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb;create=true;user=tquist;password=badpassword';
ERROR 08004: Connection authentication failure occurred. Reason: Invalid authentication..
```

Note that if you run Java DB under a Java security manager, your policy file will need to grant Java DB the privilege to connect to the LDAP server. To see how to do this, please consult the policy file used later on in Appendix F.

Appendix D: Configuring Coarse-grained Authorization

Once you have set up Authentication, you can restrict the powers of individual users. You can do this in two ways:

1. **Coarse-grained checks** – Here the Database Owner divides the application's users into two groups. One group has full authority to read and write all data. The other group merely has permission to read data.
2. **Fine-grained checks** – Here the Database Owner and individual users issue SQL GRANT/REVOKE statements to declare who can read or write specific pieces of data and who can exercise specific application functions.

This appendix describes how to setup coarse-grained checks.

You manipulate coarse-grained access by using the builtin procedure *SYSCS_SET_DATABASE_PROPERTY* to set the database properties *derby.database.fullAccessUsers* and *derby.database.readOnlyAccessUsers*. Here is an example of how to do this. The examples in this appendix assume that you are running *ij* with the *derby.properties* that you constructed in Appendix C in order to enable credentials checking. First set the read/write and read-only users. For extra security, you should configure the *derby.database.propertiesOnly* property so that users cannot override database behavior via system-wide properties specified on the command line or in the *derby.properties* file. That means that you need to copy the authentication properties from *derby.properties* inside the database by configuring them as database properties too:

```
java org.apache.derby.tools.ij
ij version 10.4

ij> connect 'jdbc:derby:testdb;create=true;user=tquist;password=tquist';

ij> --

-- Prevent our authentication settings from being overridden on the
-- command line or in derby.properties.

--

call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY

( 'derby.database.propertiesOnly','true');

0 rows inserted/updated/deleted

ij> --

-- Now that we have stated that properties cannot be overridden,
-- we need to move the authentication settings inside the database.

--
```

```
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.connection.requireAuthentication','true');
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.server', 'ldap://127.0.0.1:1389' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.provider', 'LDAP' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchAuthPW', 'ldap_pwd' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchAuthDN', 'cn=Directory Manager' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchBase', 'dc=example,dc=com' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchFilter', 'objectClass=person' );
0 rows inserted/updated/deleted
ij> --
-- OK, now we can configure read/write and read-only users.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.fullAccessUsers', 'tquist,mchrysta' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.readOnlyAccessUsers', 'thardy,jhallett' );
0 rows inserted/updated/deleted
ij> create table tquist.t1( a varchar( 20 ) );
0 rows inserted/updated/deleted
```

```
ij> insert into tquist.t1( a ) values ( 'tquist' );  
1 row inserted/updated/deleted
```

Next verify that a read/write user has those powers:

```
java org.apache.derby.tools.ij  
ij version 10.4  
ij> connect 'jdbc:derby:testdb;user=mchrysta;password=mchrysta';  
ij> insert into tquist.t1( a ) values ( 'mchrysta' );  
1 row inserted/updated/deleted  
ij> select * from tquist.t1;  
A  
-----  
tquist  
mchrysta  
  
2 rows selected
```

Finally, verify that a read-only user can read data but not write it:

```
java org.apache.derby.tools.ij  
ij version 10.4  
ij> connect 'jdbc:derby:testdb;user=thardy;password=thardy';  
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY  
( 'derby.database.fullAccessUsers', 'thardy' );  
ERROR 25502: An SQL data change is not permitted for a read-only connection, user or database.  
ij> insert into tquist.t1( a ) values ( 'thardy' );  
ERROR 25502: An SQL data change is not permitted for a read-only connection, user or database.  
ij> select * from tquist.t1;  
A  
-----  
tquist  
mchrysta
```

2 rows selected

Appendix E: Configuring Fine-grained Authorization

Now let us create another database to show how you can use fine-grained checks to restrict access to specific pieces of data. You can use fine-grained checks by themselves or in conjunction with the coarse-grained checks discussed above. In the following example, we will create two tables. One table anyone can view. The other table can only be viewed by specific users. Fine-grained checks, like coarse-grained checks, require that we run Java DB with authentication turned on. First create a database and turn on the fine-grained checks, using `ij` and the same `derby.properties` which we used above in Appendix C. As with coarse-grained checks, for added protection we copy the authentication settings into the database so that they cannot be overridden. To enable fine-grained checks, we set the `derby.database.sqlAuthorization` property:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb2:create=true;user=tquist;password=tquist';
ij> --
-- Prevent our authentication properties from being overridden on the
-- command line or in derby.properties.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.propertiesOnly','true');
0 rows inserted/updated/deleted
ij> --
-- Now that we have stated that properties cannot be overridden,
-- we need to move the authentication settings inside the database.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.connection.requireAuthentication','true');
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.server', 'ldap://127.0.0.1:1389' );
0 rows inserted/updated/deleted
```

```
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.provider', 'LDAP' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchAuthPW', 'ldap_pwd' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchAuthDN', 'cn=Directory Manager' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchBase', 'dc=example,dc=com' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchFilter', 'objectClass=person' );
0 rows inserted/updated/deleted
ij> --
-- OK, enable fine-grained authorization checks.
--
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
(
  'derby.database.sqlAuthorization',
  'true'
);
0 rows inserted/updated/deleted
```

By exiting the *ij* tool, we implicitly shut down the database. The next time the database boots up, it will enforce fine-grained authorization checks. Once enabled, these checks cannot be disabled.

Next, another user creates two tables and uses the GRANT command to let all users view one of the tables. In addition, he restricts viewing privileges on the other table:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb2;user=mchrysta;password=mchrysta';
```

```
ij> create table publicTable( a int );
0 rows inserted/updated/deleted
ij> create table restrictedTable( a int );
0 rows inserted/updated/deleted
ij> insert into publicTable( a ) values ( 1 );
1 row inserted/updated/deleted
ij> insert into restrictedTable( a ) values( 100 );
1 row inserted/updated/deleted
ij> grant select on publicTable to public;
0 rows inserted/updated/deleted
ij> grant select on restrictedTable to thardy;
0 rows inserted/updated/deleted
```

Now verify that thardy can view both tables:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb2;user=thardy;password=thardy';
ij> select * from mchrysta.publicTable;
A
-----
1

1 row selected
ij> select * from mchrysta.restrictedTable;
A
-----
100

1 row selected
```

However, other users can only view the public table:

```
java org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby:testdb2;user=jhallett;password=jhallett';
ij> select * from mchrysta.publicTable;
A
-----
1

1 row selected
ij> select * from mchrysta.restrictedTable;
ERROR 42502: User 'JHALLETT' does not have select permission on column 'A' of table 'MCHRYSTA'.'RESTRICTEDTABLE'.
```

You can also use the GRANT command to restrict write access to your tables, to control who executes your functions and procedures, to limit who add triggers to your tables, and to limit who creates foreign keys referencing your tables.

Appendix F: Configuring Java Security

The Java security manager lets you reduce the damage that your application can do. Using a security manager, the System Administrator can restrict how an application cooperates with other applications running in the same VM or elsewhere on the same machine. When you run Java DB under a security manager, you can restrict the following:

1. **Backups** – You control where the engine writes and reads database backup files.
2. **Imports and exports** – You control where the engine imports data from and where it exports data to.
3. **Jar files** – You control where the engine gets jar files of customer-coded functions and procedures.
4. **Sockets** – You control what machines can connect to the server and vice-versa.

An in-depth explanation of the Java security manager falls outside the scope of this white paper. For more information, please read the [Java 2 Platform Security Architecture paper](http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html) (<http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html>). To take advantage of these powerful

controls, first customize a copy of the template security policy file shipped with Java DB. That template lives in the Java DB distribution at *demo/templates/server.policy*. You will want to edit this template as follows:

1. **URLs** – Replace the `${derby.install.url}` variables with an **url** pointing to the directory which holds the Java DB jar files. E.g.: <file:///Users/me/javadb/lib/>.
2. **System home** – Look for instances of the string `${derby.system.home}`. Replace them with the name of the directory which holds your *derby.properties* file.
3. **Tracing** – Grant Java DB the power to manage a directory tree which will hold server trace information. Look for the `{derby.drda.traceDirectory}` variable and replace it with the directory where the server should write its diagnostic traces. For more information on Java DB tracing, please consult the section titled “Controlling tracing by using the trace facility” in Java DB's *Server and Administration Guide*.
4. **Backups/imports/jars** – Look for the first instance of the string `<<ALL FILES>>`. Make two more copies of this line so that there are three copies of this line in the file. On the first line, replace `<<ALL FILES>>` with the directory tree which you will use for holding database backups. E.g.: `/Users/me/backups`. Similarly, on the second line, replace `<<ALL FILES>>` with the directory tree which you will use for staging imported/exported data. On the third line, replace `<<ALL FILES>>` with the directory tree which holds the jar files of functions and procedures which you will load into databases. For the detailed syntax of these lines, see the javadoc for [java.io.FilePermission](#).
5. **Sysinfo** – The *sysinfo* logic appears redundantly in several Java DB jar files: *derby.jar*, *derbynet.jar*, *derbyclient.jar*, and *derbytools.jar*. If you want to run the *sysinfo* command, you will need to grant permissions to the first of these jar files which appears in your classpath. Note that if you are running Java DB via `java -jar derbyrun.jar`, then *derby.jar* is the first relevant jar file in the classpath. The template policy file grants these permissions to *derbynet.jar*. Note that the template policy file grants *sysinfo* the privilege to read all files in the system. You can restrict this to just the directory which contains the jar files wired into your classpath.
6. **Sockets** – The template policy file accepts connection requests from all hosts. You may want to restrict the template file's `java.net.SocketPermission` to just connections from a particular subdomain. For details, see the javadoc for [SocketPermission](#). In addition, you must grant *derby.jar* the privilege to connect to the LDAP server for authentication.

Here is a sample, customized policy file:

```
//
// This template policy file gives examples of how to configure the
// permissions needed to run a Derby network server with the Java
// Security manager.
//
grant codeBase "file:///Users/me/javadb/lib/derby.jar"
{
//
```

```
// These permissions are needed for everyday, embedded Derby usage.
//
permission java.lang.RuntimePermission "createClassLoader";
permission java.util.PropertyPermission "derby.*", "read";
permission java.util.PropertyPermission "user.dir", "read";
permission java.util.PropertyPermission "derby.storage.jvmInstanceId",
    "write";
permission java.io.FilePermission "/Users/me/derby/dummy","read";
permission java.io.FilePermission "/Users/me/derby/dummy${}-", "read,write,delete";

//
// This permission lets a DBA reload the policy file while the server
// is still running. The policy file is reloaded by invoking the
// SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY() system procedure.
//
permission java.security.SecurityPermission "getPolicy";

//
// This permission lets you backup and restore databases
// to and from arbitrary locations in your file system.

//
// You may want to restrict this access to specific directories.
//

// backups
permission java.io.FilePermission "/Users/me/derby/dummy/backups/-", "read,write,delete";

// imports/exports
permission java.io.FilePermission "/Users/me/derby/dummy/imports/-", "read,write,delete";

// jar files of user-written functions and procedures
```

```
permission java.io.FilePermission "/Users/me/derby/dummy/jars/-", "read,write,delete";

//

// Permissions needed for JMX based management and monitoring, which is only
// available for JVMs supporting "platform management", that is J2SE 5.0 or better.
//

// Allows this code to create an MBeanServer:
//

permission javax.management.MBeanServerPermission "createMBeanServer";
//

// Allows access to Derby's built-in MBeans, within the domain org.apache.derby.
// Derby must be allowed to register and unregister these MBeans.
// It is possible to allow access only to specific MBeans, attributes or
// operations. To fine tune this permission, see the javadoc of
// javax.management.MBeanPermission or the JMX Instrumentation and Agent
// Specification.
//

permission javax.management.MBeanPermission "org.apache.derby.*[org.apache.derby:*]","registerMBean,unregisterMBean";
//

// Trusts Derby code to be a source of MBeans and to register these in the MBean server.
//

permission javax.management.MBeanTrustPermission "register";

// This permission is needed to connect to the LDAP server in order
// to authenticate users.
//

permission java.net.SocketPermission "127.0.0.1:1389", "accept,connect,resolve";
};

grant codeBase "file:///Users/me/javadb/lib/derbynet.jar"
{
```

```
//  
  
// This permission lets the Network Server manage connections from clients.  
  
//  
  
// needed so that clients can connect to the server  
permission java.net.SocketPermission "localhost:0-", "accept";  
  
//  
  
// Needed for server tracing.  
  
//  
permission java.io.FilePermission "/Users/me/derby/dummy/traces${}-", "read,write,delete";  
  
//  
  
// Needed by sysinfo. The file permission is needed to  
// check the existence of jars on the classpath. You can  
// limit this permission to just the locations which hold  
// your jar files.  
  
//  
// In this template file, this block of permissions is granted  
// to derbynet.jar under the assumption that derbynet.jar is  
// the first jar file in your classpath which contains the  
// sysinfo classes. If that is not the case, then you will want  
// to grant this block of permissions to the first jar file  
// in your classpath which contains the sysinfo classes.  
// Those classes are bundled into the following Derby  
// jar files:  
  
//  
// derbynet.jar  
  
// derby.jar  
  
// derbyclient.jar  
  
// derbytools.jar
```

```
//  
permission java.util.PropertyPermission "user.*", "read";  
permission java.util.PropertyPermission "java.home", "read";  
permission java.util.PropertyPermission "java.class.path", "read";  
permission java.lang.RuntimePermission "getProtectionDomain";  
permission java.io.FilePermission "/Users/me/javadb/lib/-", "read";  
};
```

Now bring up the network server with a security manager and this customized policy file. As before, configure authentication using the *derby.properties* we created in Appendix C:

```
java -Djava.security.manager -Djava.security.policy=/opt/DerbyTrunk/./server.policy \  
org.apache.derby.drda.NetworkServerControl start -p 8246
```

Shut down the network server so that we can move on to the final set of examples, which demonstrate all of the security mechanisms working together:

```
java org.apache.derby.drda.NetworkServerControl shutdown -p 8246 -user tqvist -password tqvist
```

Appendix G: Putting It All Together

In this appendix we show how to enable all Java DB defenses. Here we again use the *derby.properties* from Appendix C, which enables LDAP-based authentication. We also use the security policy devised in Appendix F. First bring up the server, turning on SSL and Java Security:

```
java -Djavax.net.ssl.keyStore=/Users/me/vault/ServerKeyStore \  
-Djavax.net.ssl.keyStorePassword=secretServerPassword \  
-Djavax.net.ssl.trustStore=/Users/me/vault/ServerTrustStore \  
-Djavax.net.ssl.trustStorePassword=secretServerTrustStorePassword \  
-Djava.security.manager \  
-Djava.security.policy=/opt/DerbyTrunk/./server.policy \  
org.apache.derby.drda.NetworkServerControl start -p 8246 -ssl peerAuthentication
```

Now the Database Owner creates an encrypted database, turns on both coarse-grained and fine-grained authorization, and creates some data which everyone can read but only he can write:

```

java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
    -Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
    -Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
    -Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.tools.ij
ij version 10.4
ij> connect
'jdbc:derby://localhost:8246/mchrystaEncryptedDB;create=true;user=mchrysta;password=mchrysta;dataEncryption=true;encryptionAlgorithm=
Blowfish/CBC/NoPadding;bootPassword=mySuperSecretBootPassword;ssl=peerAuthentication';
ij> --
-- Prevent our authentication properties from being overridden on the
-- command line or in derby.properties.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.propertiesOnly','true');
Statement executed.
ij> --
-- Now that we have stated that properties cannot be overridden,
-- we need to move the authentication settings inside the database.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.connection.requireAuthentication','true');
Statement executed.
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.server', 'ldap://127.0.0.1:1389' );
Statement executed.
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.provider', 'LDAP' );
Statement executed.
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.ldap.searchAuthPW', 'ldap_pwd' );

```

Statement executed.

```
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.Idap.searchAuthDN', 'cn=Directory Manager' );
```

Statement executed.

```
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.Idap.searchBase', 'dc=example,dc=com' );
```

Statement executed.

```
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.authentication.Idap.searchFilter', 'objectClass=person' );
```

Statement executed.

```
ij> --
```

```
-- Turn on coarse-grained authorization
```

```
--
```

```
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.fullAccessUsers', 'tquist,mchrysta' );
```

Statement executed.

```
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.readOnlyAccessUsers', 'thardy,jhallett' );
```

Statement executed.

```
ij> --
```

```
-- Turn on fine-grained authorization
```

```
--
```

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
(
```

```
    'derby.database.sqlAuthorization',
```

```
    'true'
```

```
);
```

Statement executed.

```
ij> --
```

```
-- Shutdown the database and bring it back up. This will
```

```
-- cause the authorization property to be applied.
```

```

--
connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;shutdown=true;user=mchrysta;password=mchrysta;ssl=peerAuthentication';
ERROR 08006: DERBY SQL error: SQLCODE: -1, SQLSTATE: 08006, SQLERRMC: Database 'mchrystaEncryptedDB' shutdown.
ij> connect
'jdbc:derby://localhost:8246/mchrystaEncryptedDB;user=mchrysta;password=mchrysta;bootPassword=mySuperSecretBootPassword;ssl=peer
Authentication';
ij(CONNECTION1)> --
-- Create some data and let everyone see it.
--
create table mchrysta.t1( a varchar( 20 ) );
0 rows inserted/updated/deleted
ij(CONNECTION1)> insert into mchrysta.t1( a ) values ( 'mchrysta' );
1 row inserted/updated/deleted
ij(CONNECTION1)> grant select on table mchrysta.t1 to public;
0 rows inserted/updated/deleted

```

Verify that another user can read the newly created data but not write it:

```

java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.tools.ij
ij version 10.4
ij> connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;user=tquist;password=tquist;ssl=peerAuthentication';
ij> --
-- Verify that this user can see the data.
--
select * from mchrysta.t1;
A
-----
mchrysta

1 row selected
ij> --

```

```
-- ...but not write the data
```

```
--
```

```
insert into mchrysta.t1( a ) values ( 'tquist' );
```

```
ERROR 42500: User 'TQUIST' does not have insert permission on table 'MCHRYSTA'.T1'.
```

Now bring down the server:

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \  
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \  
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \  
-Djavax.net.ssl.keyStorePassword=secretClientPassword \  
org.apache.derby.drda.NetworkServerControl shutdown -p 8246 \  
-user mchrysta -password mchrysta \  
-ssl peerAuthentication
```

Glossary

attacker – A person or organization which seeks to compromise the security of a system.

damages – The harm done to a system by an attacker. Includes denial-of-service, theft of secrets, and corruption of data.

database owner – The person who creates a database and configures its security.

insider – An attacker, such as a disgruntled co-worker, who operates inside the firewall and enjoys the presumption of friendliness.

malware – A program which compromises security, such as a virus, worm, or spider.

outsider – An attacker who operates outside the firewall.

system administrator – The account which launches Java DB and is responsible for configuring the security of the Java DB system.

technique – A mechanism for compromising the security of a system, such as man-in-the-middle or SQL injection.

user – A person authorized to use a Java DB application.

vulnerability – A feature of Java DB which attackers can exploit in order to cause damage.

