SQL Injection

• A SQL injection vulnerability is caused when unsanitized user inputs are used to create SQL statements

    String sql = "SELECT * FROM users WHERE username = '" + name + "' AND password = '" + pass + "'";

    ResultSet rs = stmt.executeQuery(sql);

• Could allow an attacker to run arbitrary SQL code
SQL Injection

• Entering something like "Robert'; DROP TABLE users; --" for the username (and "x" for the password) would give us

```
SELECT * FROM users WHERE username = 'Robert'; DROP TABLE users;
--' AND password = 'x'
```

Note that "--" (two dashes) typically starts a comment in SQL.
Sanitizing Inputs

• Using a PreparedStatement instead of concatenating strings and using Statement can protected against SQL injection, but only if used properly
Sanitizing Inputs

- This

    String sql = "SELECT * FROM users WHERE username = '" + name + "' AND password = '" + pass + "'";

    PreparedStatement stmt = new PreparedStatement(sql);

    ResultSet rs = stmt.executeQuery();

- Is no safer than this

    String sql = "SELECT * FROM users WHERE username = '" + name + "' AND password = '" + pass + "'";

    Statement stmt = new Statement();

    ResultSet rs = stmt.executeQuery(sql);
PreparedStatement

• Instead do this

```java
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM users WHERE username = ? AND password = ?"
);

stmt.setString(1, name);
stmt.setString(2, pass);

ResultSet rs = stmt.executeQuery();
```
ResultSets

• By default, PreparedStatements (and Statements) create ResultSets that are FORWARD ONLY
  – This means that they can only be iterated over once
    • Calls to ResultSet.previous() or ResultSet.first() will throw an exception
A common pattern if more flexible access is needed is to read the results into an `ArrayList` of arrays or `ArrayList` of `HashMaps`.

```java
ResultSet rs = stmt.executeQuery();
ResultSetMetaData md = rs.getMetaData();
int columnCount = md.getColumnCount();
ArrayList<Object[]> results = new ArrayList<Object[]>();
while (rs.next()) {
    Object[] row = new Object[columnCount];
    for (int i = 0; i < columnCount; ++i) {
        row[i] = rs.getObject(i + 1);
    }
    results.add(row);
}
// close rs and use results
```
ResultSet rs = stmt.executeQuery();
ResultSetMetaData md = rs.getMetaData();
int columnCount = md.getColumnCount();
ArrayList<HashMap<String, Object>> results =
    new ArrayList<HashMap<String, Object>>();
while (rs.next()) {
    HashMap<String, Object> row = new HashMap<String, Object>();
    for (int i = 1; i <= columnCount; ++i) {
        row.put(md.getColumnLabel(i), rs.getObject(i));
    }
    results.add(row);
}
// close rs and use results

• But this can be inefficient for large data sets
Scrollability

- ResultSets can be made *scrollable* for random access
  - Must be supported by the DBMS-specific driver
    - Supported by most drivers
- Three types:
  - ResultSet.TYPE_FORWARD_ONLY
    - Movement can only be forward (default)
  - ResultSet.TYPE_SCROLL_INSENSITIVE
    - Movement can be random, not sensitive to changes to underlying dataset
  - ResultSet.TYPE_SCROLL_SENSITIVE
    - Movement can be random, may reflect changes to underlying dataset
Scrollability

• If a ResultSet's type is *scroll sensitive* then it *may* reflect changes to the underlying dataset
  – Depends on many factors
    • Support by DBMS, transaction isolation level, whether entire dataset is fetched in one call or multiple calls, etc.
  – Generally, should not depend on this behavior working
Scrollability

• The type of the ResultSet must be specified when creating the Statement (*not* when creating the ResultSet)

```java
PreparedStatement stmt = conn.prepareStatement(
    sql,
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_AT_COMMIT);

...

ResultSet rs = stmt.executeQuery();
```
Concurrency

• Theoretically, ResultSets can be updated
  – But support for this varies by DMBS
• Two options:
  – ResultSet.CONCUR_READ_ONLY
    • The values of a ResultSet are read-only (default)
  – ResultSet.CONCUR_UPDATABLE
    • The values of a ResultSet can be changed via ResultSet.updateObject() (or updateString(), updateFloat(), etc.) and flushed back to the database with updateRow()
• Generally, I suggest not relying on this functionality
Holdability

• Sometimes it is possible to keep using a ResultSet after a commit
  – Support varies by DBMS and other factors
• Two options:
  – ResultSet.CLOSE_CURSORS_AT_COMMIT
    • The ResultSet is closed on a commit and cannot be used after (default)
  – ResultSet.HOLD_CURSORS_OVER_COMMIT
    • The values of the ResultSet can be accessed after a commit
• Generally, I suggest *not* relying on this functionality
Limited Use

• In general, I would not use
  • TYPE_SCROLL_SENSITIVE
  • CONCUR_UPDATABLE
  • HOLD_CURSORS_OVER_COMMIT
    – Too many situations where it doesn't work
    – Can interfere with transaction isolation

• However, TYPE_SCROLL_INSENSITIVE can be useful
Data Sources

• Using DriverManager.getConnection() is only recommended for simple applications

• A pooled DataSource is recommended
  – Most DBMSs ship with simple DataSource implementations
    • Generally just proof-of-concept
  – More robust DataSource implementations are recommended
    • e.g., C3P0
Data Sources

• Data sources are generally configured via an application server and added to a Java naming service

• Can also be configured directly for standalone applications (that don't run in an application server)
Data Sources

• C3P0 example:

```java
String url = "jdbc:postgresql://" + host + ":" + port + "/" + database;

ComboPooledDataSource source = new ComboPooledDataSource();
source.setDriverClass("org.postgresql.Driver");
source.setJdbcUrl(url);
source.setUser(username);
source.setPassword(password);

Connection conn = source.getConnection();

...

conn.close();  // doesn't actually close, returns the connection to the pool

...

DataSources.destroy(source);
```
Data Sources

• In general, an application should create one DataSource and use that single DataSource through the lifetime of the application
  – Application context or Singleton pattern
  – Multiple data sources can be used to connect to multiple databases, but generally want one data source per database
JdbcRowSet

- JdbcRowSet objects can be created from ResultSet objects
  - Underlying ResultSet must be scrollable

```java
PreparedStatement stmt = conn.prepareStatement(
    sql,
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_AT_COMMIT);

ResultSet rs = stmt.executeQuery();
JdbcRowSet jdbcRs = new JdbcRowSetImpl(rs);
```
JdbcRowSet

• JdbcRowSets can be used as JavaBeans components
  – Can be used by tools that work with JavaBeans
  – Uses the JavaBeans notification mechanism
CachedRowSet

• Row sets can be *disconnected* from the underlying data source
  – A CachedRowSet can be initialized with a data source name
    • Requires a Java naming service

    ```java
    CachedRowSet crs = new CachedRowSetImpl();
    crs.setDataSourceName(name);
    ```

  – Simple applications can set the connection properties directly on the CachedRowSet object

    ```java
    CachedRowSet crs = new CachedRowSetImpl();
    crs.setUrl(url);
    crs.setUsername(username);
    crs.setPassword(password);
    ```
CachedRowSet

- SQL commands are executed by setting the command on the CachedRowSet
  
  crs.setCommand(sql);
  crs.execute();

- A CachedRowSet
  - Executes a command
  - Populates itself from the data
  - Disconnects from the data source
  - Allows you to still access the data
CachedRowSets

- Have rarely seen CachedRowSets (or JdbcRowSets) used in actual production
- Instead, 3 options:
  - Just use ResultSet
    - But this keeps the transaction open
  - Copy the ResultSet data into ArrayList/arrays/HashMaps
  - Create objects from the ResultSet data