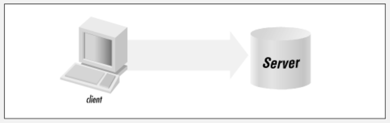
Database Applications

Database Applications

- the database provides data storage
- the database application does something with the data

- the role of the database varies somewhat depending on the type of application
  - for a Java application, database provides storage across application instances
    - data can be stored in memory while the program runs; the database provides persistent storage and allows sharing between separate instances
  - for a web application, database provides storage across operations
    - HTTP is a stateless protocol, though sessions allow for persistence between requests

Two-Tier Architecture



O'Reilly, MySQL & mSQL

The database is the *server*.

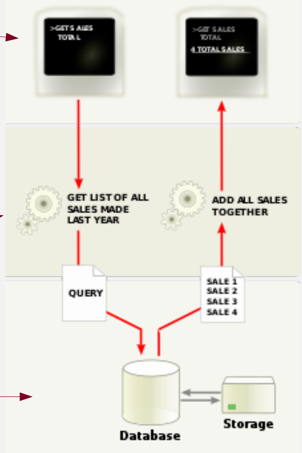
- stores data
- executes queries

The application that uses the database is the *client*.

- does everything else – UI/presentation, sends queries/processes results, carries out business logic

Three-Tier Architecture

- *presentation tier* provides the user interface
  - handles user interaction
  - displays output
  - acquires input
  - commonly implemented over the web, in which case it is also known as the *web server tier*
- *middle* or *application tier* provides the application logic
  - handles the rules, constraints, logic of the program
  - can integrate data from multiple sources
  - processes data to provide a response to the client
- *data management tier* stores the data
  - typically a database, in which case it is also known as the *database tier*
  - stores data
  - executes queries



https://en.wikipedia.org/wiki/Multitier_architecture

Three-Tier Architecture

Advantages.

- heterogeneous systems
  - allows different platforms, technologies, and components within each layer
- modularity
  - presentation (UI), business logic, and data management are separate
  - each part can be developed, maintained, and tested independently
- integrated data access
  - application layer (middle tier) allows for centralized management of data and transparent handling of multiple database systems
- scalability
  - tiers can be scaled independently as demand dictates
- thin clients
  - client is responsible only for presentation (UI)
  - reduces the hardware requirements for clients, allowing more devices access
  - minimizes client-side configuration and management

Technologies

two-tier

- client-server architecture
- in this course
  - database (server) – MySQL
  - client – Java application
  - utilizes free and open-source software (*)

(*) not necessarily free for all uses e.g. commercial use

three-tier

- a traditional approach for web applications is LAMP
  - OS – **L**inux
  - presentation tier – **A**pache (webserver), HTML, CSS
  - data management tier – **M**ySQL (DBMS)
  - application tier – **P**HP (server-side scripting and programming language)
  - utilizes free and open-source software (*)
- variants for other operating systems (WAMP, MAMP) and can substitute other components

Communicating With Databases

For both web and Java applications, communication with the database is done through a library of routines.

- the library is particular to both the application programming language and the database vendor
  - for PHP and MySQL, we are using MySQLi
  - Java provides a standard API (JDBC) with different backends for different vendors (we are using the MySQL one)

Communicating With Databases

Steps –

- establish a connection
  - to a particular database, host, and port as a particular user
- send queries
  - one-time queries without user input
  - queries incorporating user input
  - repeated similar queries
- process results
- close connection

can be repeated using the same connection

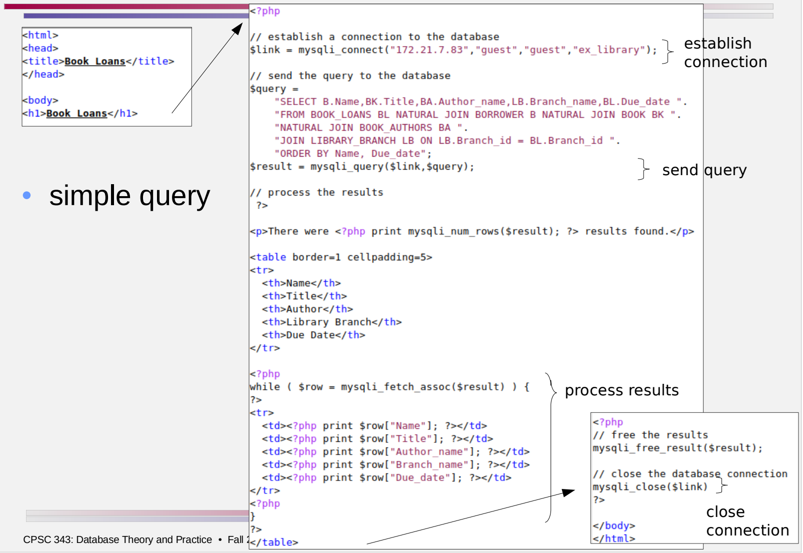
- error checking
  - should be done at each step in the process
  - should avoid publicly revealing details of the database schema in error messages (not useful to users, can be helpful to attackers)

Key Points

- how to do each of the steps (including error handling)
  - depends on the particular library / API
- prepared statements
- security
  - user accounts
  - handling passwords
  - SQL injection and other attacks

Web – HTML/PHP

- simple query



```
<html>
<head>
<title>Book Loans</title>
</head>

<body>
<h1>Book Loans</h1>
```

```
<?php

// establish a connection to the database
$link = mysqli_connect("172.21.7.83","guest","guest","ex_library");

// send the query to the database
$query =
    "SELECT B.Name,BK.Title,BA.Author_name,LB.Branch_name,BL.Due_date ".
    "FROM BOOK_LOANS BL NATURAL JOIN BORROWER B NATURAL JOIN BOOK BK ".
    "NATURAL JOIN BOOK_AUTHORS BA ".
    "JOIN LIBRARY_BRANCH LB ON LB.Branch_id = BL.Branch_id ".
    "ORDER BY Name, Due_date";
$result = mysqli_query($link,$query);

// process the results
 ?>

<p>There were <?php print mysqli_num_rows($result); ?> results found.</p>

<table border=1 cellpadding=5>
<tr>
  <th>Name</th>
  <th>Title</th>
  <th>Author</th>
  <th>Library Branch</th>
  <th>Due Date</th>
</tr>

<?php
while ( $row = mysqli_fetch_assoc($result) ) {
?>
<tr>
  <td><?php print $row["Name"]; ?></td>
  <td><?php print $row["Title"]; ?></td>
  <td><?php print $row["Author_name"]; ?></td>
  <td><?php print $row["Branch_name"]; ?></td>
  <td><?php print $row["Due_date"]; ?></td>
</tr>
<?php
}
?>
</table>
```

```
<?php
// free the results
mysqli_free_result($result);

// close the database connection
mysqli_close($link)
?>

</body>
</html>
```

establish connection

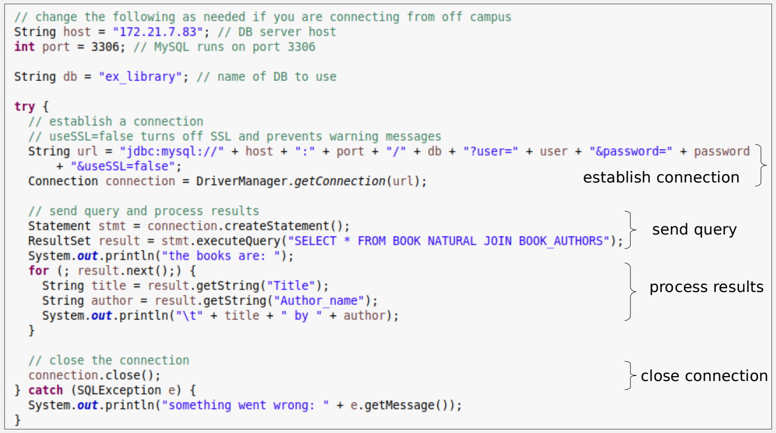
send query

process results

close connection

Java – JDBC

- simple query



```
// change the following as needed if you are connecting from off campus
String host = "172.21.7.83"; // DB server host
int port = 3306; // MySQL runs on port 3306

String db = "ex_library"; // name of DB to use

try {
  // establish a connection
  // useSSL=false turns off SSL and prevents warning messages
  String url = "jdbc:mysql://" + host + ":" + port + "/" + db + "?user=" + user + "&password=" + password
      + "&useSSL=false";
  Connection connection = DriverManager.getConnection(url);

  // send query and process results
  Statement stmt = connection.createStatement();
  ResultSet result = stmt.executeQuery("SELECT * FROM BOOK NATURAL JOIN BOOK_AUTHORS");
  System.out.println("the books are: ");
  for (; result.next();) {
    String title = result.getString("Title");
    String author = result.getString("Author_name");
    System.out.println("\t" + title + " by " + author);
  }

  // close the connection
  connection.close();
} catch (SQLException e) {
  System.out.println("something went wrong: " + e.getMessage());
}
```

establish connection

send query

process results

close connection

Prepared Statements

SELECT *
FROM SAILOR NATURAL JOIN
RESERVATION
WHERE Sname='Dustin'

SELECT *
FROM SAILOR NATURAL JOIN
RESERVATION
WHERE Sname='Horatio'

When a query is sent to the DBMS –

- the text is parsed, the syntax checked, and the names validated
- a query tree is built
- an execution plan is developed
- the plan is executed

these steps depend only on the structure of the query, not specific values

using a *prepared statement* allows this work to be cached

prepared statements also provide protection against SQL injection

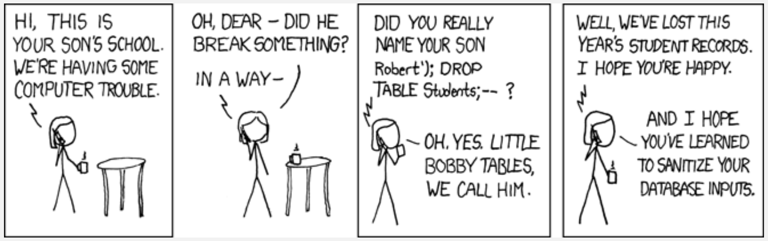
Security – User Accounts

- users authenticate to the DBMS
  - each user has a separate DBMS account
  - can leverage the DBMS security mechanisms
  - can't bypass security by bypassing application
- “one big application user” model – users authenticate to the application
  - database stores authentication information (e.g. username and password)
  - application uses a single (highly-privileged) DBMS account when it connects to the DB
  - application can manage users and privileges without needing the powerful GRANT privilege
- proxy users
  - one DBMS account per user role rather than per user
  - DBMS accounts can have more targeted privileges
  - application can manage users

Security – Password Management

- assume that passwords written down can be seen
  - e.g. values stored in hidden form elements are visible in the HTML source
  - e.g. cookies are stored in plaintext on the client side, and are sent over the network with each request
  - e.g. sessions do not guarantee that stored info is visible only to the user that created the session
- avoid storing passwords – have user enter interactively
  - don't hardcode sensitive passwords
- only store encrypted passwords
- don't store passwords in a publicly accessible place
  - use a password wallet with a single master password
  - for web applications, put passwords in a separate configuration file outside the web root (or use .htaccess to prevent access)

SQL Injection



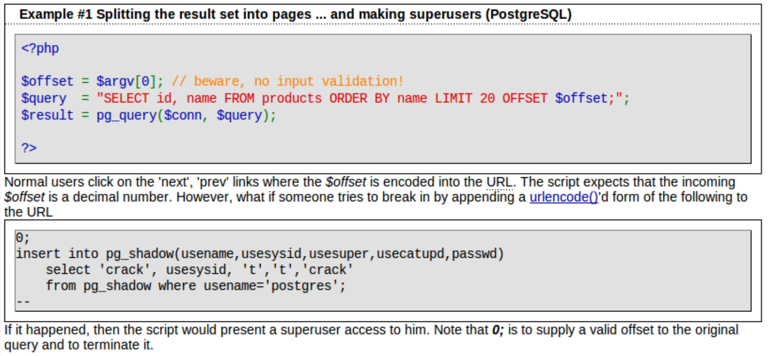
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

http://xkcd.com/327/

SQL injection is a technique an attacker can use to gain information or access through queries sent to the database.

The vulnerability comes from incorporating unchecked user inputs into a query.

SQL Injection



Example #1 Splitting the result set into pages ... and making superusers (PostgreSQL)

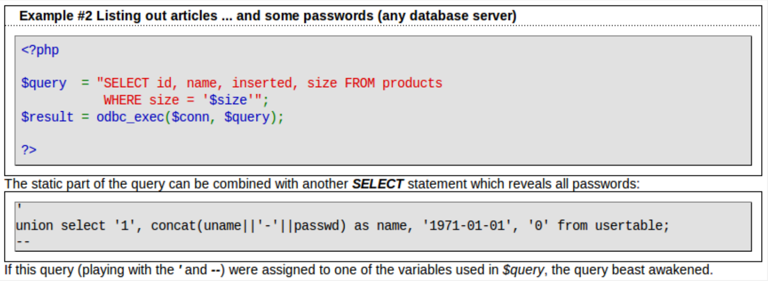
```
<?php

$offset = $argv[0]; // beware, no input validation!
$query  = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET $offset;";
$result = pg_query($conn, $query);

?>
```

Normal users click on the 'next', 'prev' links where the *$offset* is encoded into the URL. The script expects that the incoming *$offset* is a decimal number. However, what if someone tries to break in by appending a urlencode()'d form of the following to the URL

```
0;
insert into pg_shadow(usename,usesysid,usesuper,usecatupd,passwd)
    select 'crack', usesysid, 't','t','crack'
    from pg_shadow where usename='postgres';
--
```

If it happened, then the script would present a superuser access to him. Note that ***0;*** is to supply a valid offset to the original query and to terminate it.

SQL Injection



Example #2 Listing out articles ... and some passwords (any database server)

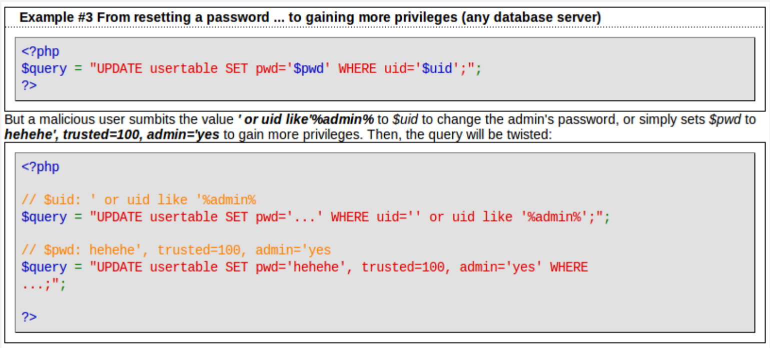
```
<?php

$query  = "SELECT id, name, inserted, size FROM products
           WHERE size = '$size'";
$result = odbc_exec($conn, $query);

?>
```

The static part of the query can be combined with another ***SELECT*** statement which reveals all passwords:

```
'
union select '1', concat(uname||'-'||passwd) as name, '1971-01-01', '0' from usertable;
--
```

If this query (playing with the **'** and **--**) were assigned to one of the variables used in *$query*, the query beast awakened.

SQL Injection



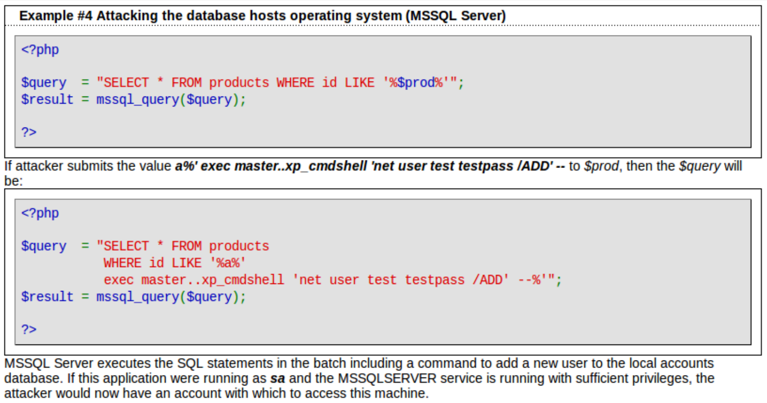
Example #3 From resetting a password ... to gaining more privileges (any database server)

```
<?php
$query = "UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';";
?>
```

But a malicious user sumbits the value ***' or uid like'%admin%*** to *$uid* to change the admin's password, or simply sets *$pwd* to ***hehehe', trusted=100, admin='yes*** to gain more privileges. Then, the query will be twisted:

```
<?php

// $uid: ' or uid like '%admin%
$query = "UPDATE usertable SET pwd='...' WHERE uid='' or uid like '%admin%';";

// $pwd: hehehe', trusted=100, admin='yes
$query = "UPDATE usertable SET pwd='hehehe', trusted=100, admin='yes' WHERE
...;";

?>
```

SQL Injection



Example #4 Attacking the database hosts operating system (MSSQL Server)

```
<?php

$query  = "SELECT * FROM products WHERE id LIKE '%$prod%'";
$result = mssql_query($query);

?>
```

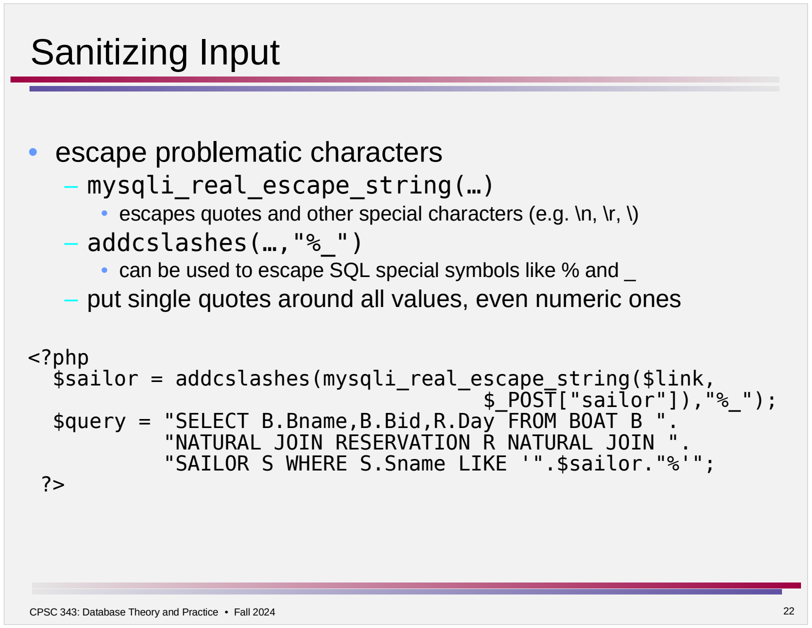
If attacker submits the value ***a%' exec master..xp_cmdshell 'net user test testpass /ADD' --*** to *$prod*, then the *$query* will be:

```
<?php

$query  = "SELECT * FROM products
           WHERE id LIKE '%a%'
           exec master..xp_cmdshell 'net user test testpass /ADD' --%'";
$result = mssql_query($query);

?>
```

MSSQL Server executes the SQL statements in the batch including a command to add a new user to the local accounts database. If this application were running as ***sa*** and the MSSQLSERVER service is running with sufficient privileges, the attacker would now have an account with which to access this machine.

Security – SQL Injection

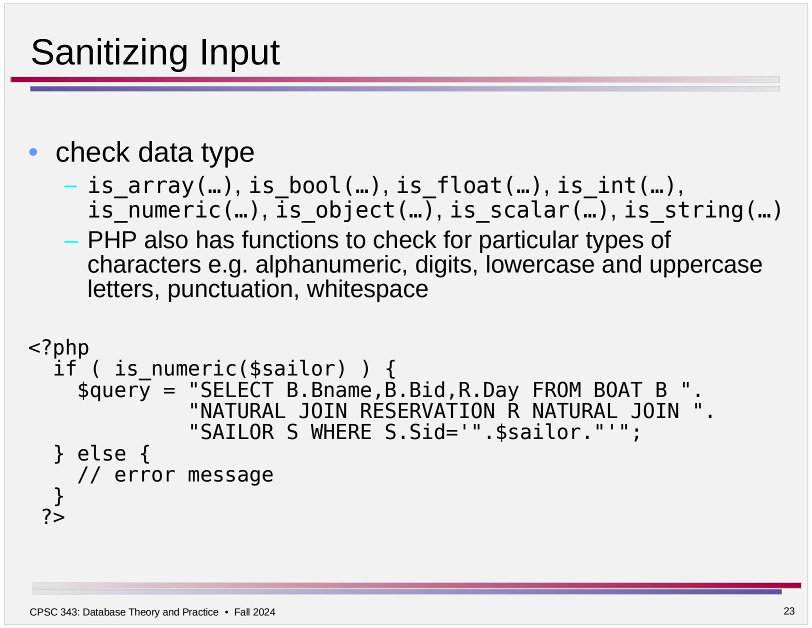
- malicious users can exploit the incorporation of raw input into queries
- to prevent
  - don't trust user input
    - use prepared statements
      - also more efficient for repeated queries with different values
    - interpret table and attribute names rather than including directly
    - validate and sanitize input
      - verify data type, escape problematic characters, put single quotes around all values (including numbers)
  - deny access
    - *principle of least privilege* – limit privileges of DB accounts
    - use views and stored routines to limit what users can see and manipulate
  - keep secrets
    - don't reveal internals of database (e.g. table names) in error messages or other public places



Sanitizing Input

- escape problematic characters
  - `mysqli_real_escape_string(…)`
    - escapes quotes and other special characters (e.g. \n, \r, \)
  - `addcslashes(…,"%_")`
    - can be used to escape SQL special symbols like % and _
  - put single quotes around all values, even numeric ones

```
<?php
  $sailor = addcslashes(mysqli_real_escape_string($link,
                                     $_POST["sailor"]),"%_");
  $query = "SELECT B.Bname,B.Bid,R.Day FROM BOAT B ".
           "NATURAL JOIN RESERVATION R NATURAL JOIN ".
           "SAILOR S WHERE S.Sname LIKE '".$sailor."%'";
 ?>
```



Sanitizing Input

- check data type
  - `is_array(…)`, `is_bool(…)`, `is_float(…)`, `is_int(…)`, `is_numeric(…)`, `is_object(…)`, `is_scalar(…)`, `is_string(…)`
  - PHP also has functions to check for particular types of characters e.g. alphanumeric, digits, lowercase and uppercase letters, punctuation, whitespace

```
<?php
  if ( is_numeric($sailor) ) {
    $query = "SELECT B.Bname,B.Bid,R.Day FROM BOAT B ".
             "NATURAL JOIN RESERVATION R NATURAL JOIN ".
             "SAILOR S WHERE S.Sid='".$sailor."'";
  } else {
    // error message
  }
 ?>
```