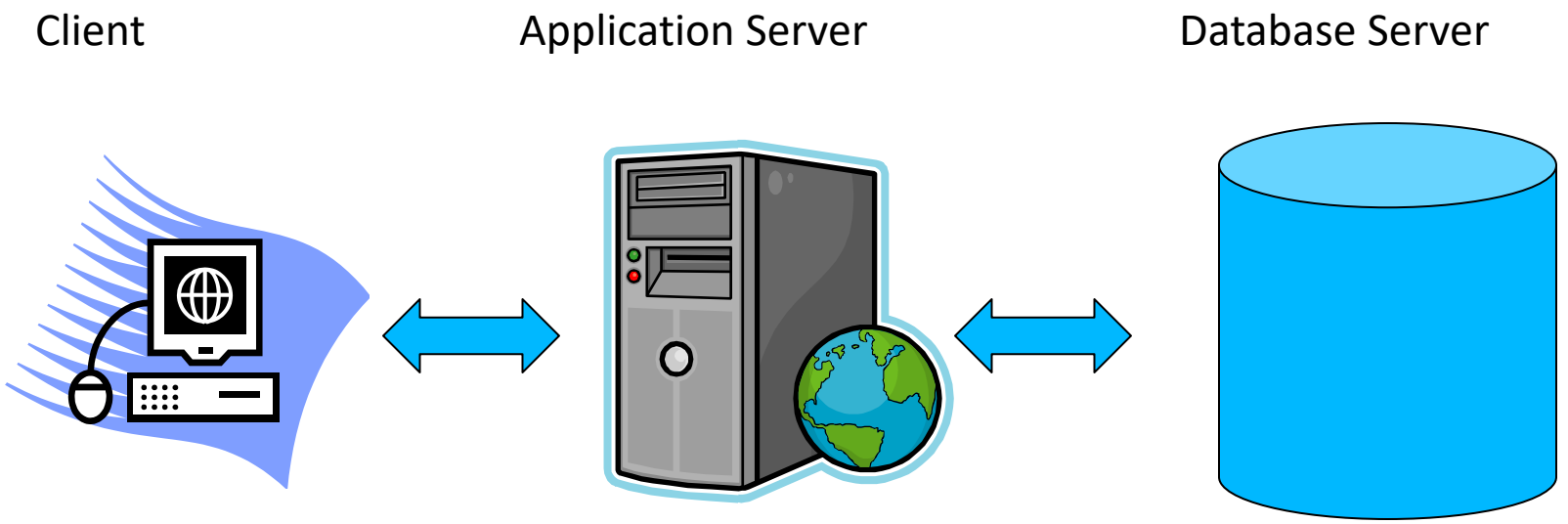
CS5331: Web Security

Lecture 4: Server-side Injection Attacks

SQL Injection

Three-Tier Architecture



- http://www.linuxjournal.com/article/3508

SQL: Background

- Read:
  - https://www.w3schools.com/sql/default.asp
- Common statements/constructs:
  - SELECT statement: to select data from a database

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND ...;
```

  - INSERT INTO statement : to insert new records in a table

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

  - UPDATE statement: to modify the existing records in a table

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

SQL: Background

- SELECT TOP clause: to specify the number of records to return

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;

SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

- UNION operator:
  to combine the result-set of two or more SELECT statements:
  - Each SELECT statement within UNION must have the same number of columns
  - The columns must also have similar data types
  - The columns in each SELECT statement must also be in the same order
- DROP DATABASE statement: to drop an existing SQL database

```
DROP DATABASE databasename;
```

SQL: Background

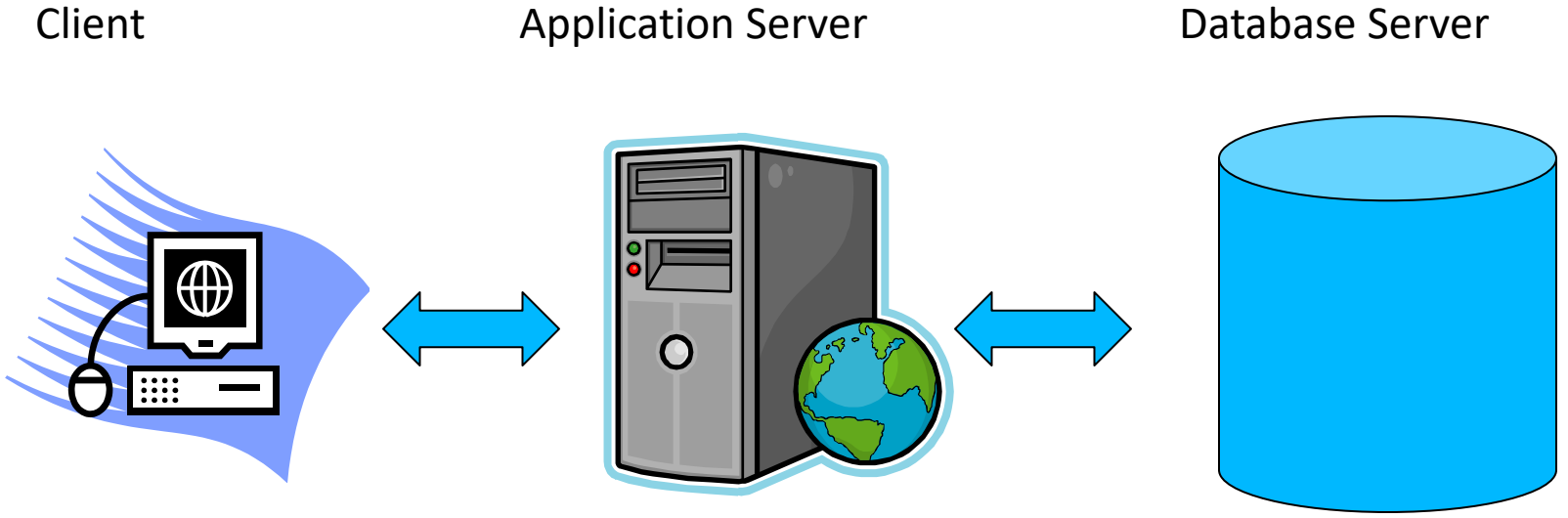
- SQL comments:
  - Single line comments:
    - Starts with #: E.g. `# A single-line comment`
    - Starts with --: E.g. `-- A single-line comment`
  - Multi line comments:
    - E.g.: `/* A multi-line comment */`

SQL: Background

```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// Create connection
$conn = mysqli_connect($servername, $username, $password, $dbname);
// Check connection
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}

$sql = "SELECT id, firstname, lastname FROM MyGuests";
$result = mysqli_query($conn, $sql);

if (mysqli_num_rows($result) > 0) {
    // output data of each row
    while($row = mysqli_fetch_assoc($result)) {
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";
    }
} else {
    echo "0 results";
}

mysqli_close($conn);
?>
```

From: https://www.w3schools.com/php/php_mysql_select.asp

SQL Example



Userid = 1234

```
SELECT * FROM user_data
WHERE user_id = 1234
```

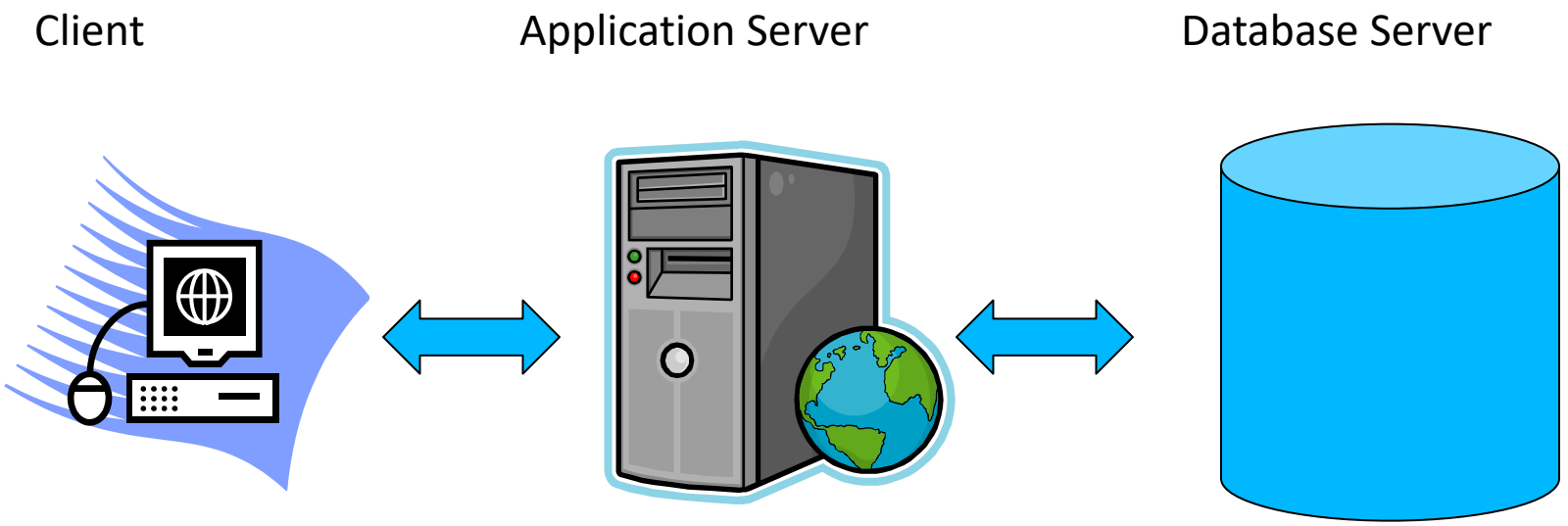
SQL Injection Attack

- A common pattern:
  - Application server gets inputs from users, creates SQL statements as strings, and sends the statements to DB server
- For example:

```
query = "SELECT * FROM user_data WHERE userid = " + userid;
```

- How to exploit this to inject SQL statements?
  - userid: 0 OR 1=1

SQL Injection Attack

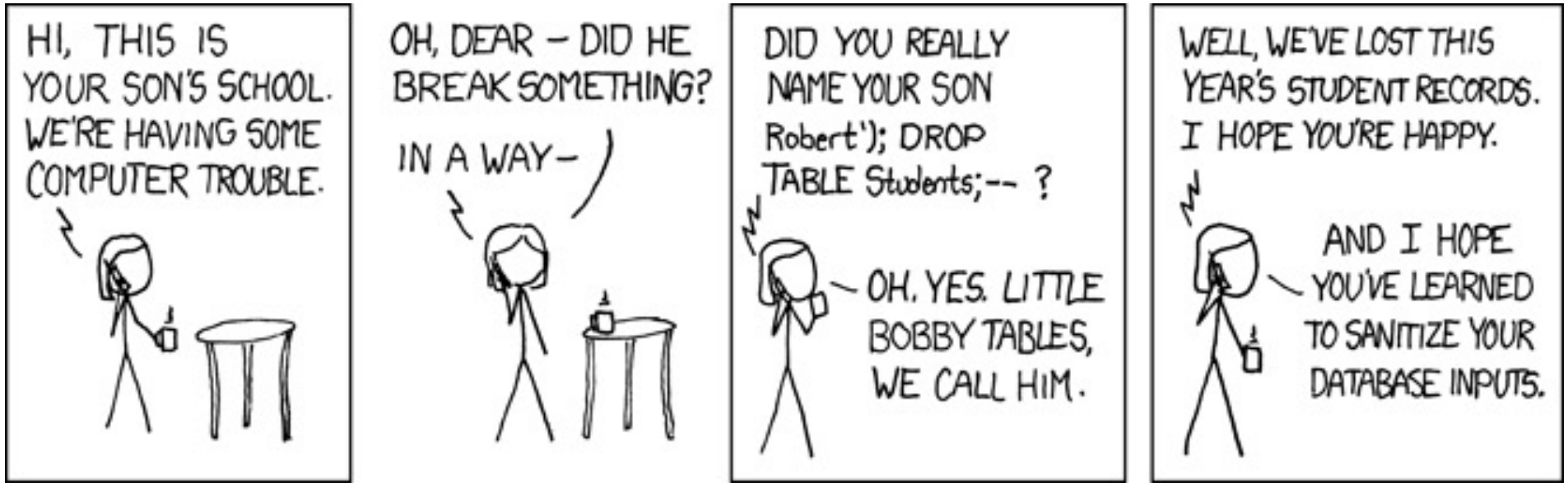


Userid = 0 OR 1=1

```
SELECT * FROM user_data
WHERE user_id = 0 OR 1=1
```

```
SELECT * FROM user_data
WHERE user_id = 0 OR 1=1;
INSERT INTO user_data
VALUES ("hacker", "passwd" …)
```

SQLI: Example ("Bobby Tables")



Source: https://xkcd.com/327/

SQLI: Rare but Still Happening

- 2016 Symantec Endpoint Protection (SEP): CSRF + SQLi
- 2015 Archos attack – leaked ≤ 100K customer details
- 2015 Joomla SQLi
- 2014 2 SQLi in Wordpress plugin
- 2014 SQLi Tesla website
- ...

SQLI: Classification

- Ref: Halfond et al., "*A Classification of SQL Injection Attacks and Countermeasures*", ISSSE, 2006
- Sample vulnerable code:

```
1.  String login, password, pin, query
2.  login = getParameter("login");
3.  password = getParameter("pass");
3.  pin = getParameter("pin");
4.  Connection conn.createConnection("MyDataBase");
5.  query = "SELECT accounts FROM users WHERE login='" +
6.          login + "' AND pass='" + password +
7.          "' AND pin=" + pin;
8.  ResultSet result = conn.executeQuery(query);
9.  if (result!=NULL)
10.     displayAccounts(result);
11. else
12.     displayAuthFailed();
```

Figure 1: Excerpt of servlet implementation.

SQLI: Classification

SQLI's Attack Payloads [Halfond et al.]:

- Preliminary/reconnaissance payloads:
  - Identifying injectable parameters
  - Performing database finger-printing
  - Determining database schema
- Exploitation payloads:
  - Bypassing authentication
  - Extracting data
  - Adding or modifying data
  - Performing denial of service
  - Executing remote commands
  - Performing privilege escalation
- Other:
  - Evading detection

SQLI: More Tricks

- There are (again) many attack vectors
  - SQL Injection cheat sheet
- Easier to get right than XSS
  - Beware of character set encoding
  - See rules for string literals (e.g. MYSQL)
  - Varies by database engine

Prepared Statements

- The *best* solution: Prepared Statements
  - A less powerful API that only does what you want: only runs the queries set in templates
  - Syntactically similar, but semantically very different
  - Properly *separates control and data channels*
  - Runs faster too:
    good for both performance and security!
  - Example (PHP): see next slides
  - Ref: https://www.w3schools.com/php/php_mysql_prepared_statements.asp

Example of Prepared Statements

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";
// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
// prepare and bind
$stmt = $conn->prepare("INSERT INTO MyGuests (firstname, lastname,
email) VALUES (?, ?, ?)");
$stmt->bind_param("sss", $firstname, $lastname, $email);
```

Example of Prepared Statements

```
// set parameters and execute
$firstname = "John";
$lastname = "Doe";
$email = "john@example.com";
$stmt->execute();

$firstname = "Mary";
$lastname = "Moe";
$email = "mary@example.com";
$stmt->execute();
...
echo "New records created successfully";

$stmt->close();
$conn->close();
?>
```

SQLI: Other Defenses

- Use the proper query invocation function:
  - `mysqli::query()`: does *not* allow multiple queries
  - `mysqli::multi_query()`: allow multiple queries (avoid this!)

- Randomize database schema

- Use the principle of least privilege:
  - Set the permissions of the database username/password as tightly as possible
  - Example:
    If the operation is only to display data (SELECT permission), then there should be no rights for INSERT/UPDATE/DROP permissions

Other Injection Attacks: OS Command Injection

Command Injection

```
www.mysite.com/viewcontent.php?filename=my_great_content.txt;ls
```

```php
<?php
echo shell_exec('cat '.$_GET['filename']);
?>
```

Runs the `ls` command on the server

Command Injection

- Most common attack goals:
  - Dump the server's password file:
    `cat /etc/passwd`
  - Add an admin user:
    - Add a user:
      `useradd new_user; passwd new_user`
    - Add a user into the admin group:
      `usermod -G admin new_user`
    - Delete an existing user:
      `getent group admin;`
      `userdel existing_user`

Command Injection: Defenses

1. Apply input validation by using a whitelist
2. Apply input escaping:
   - `escapeshellarg()`
   - "Adds single quotes around a string and quotes/escapes any existing single quotes, allowing you to pass a string directly to a shell function and having it be treated as a single safe argument"
   - (Ref: http://php.net/manual/en/function.escapeshellarg.php)
   - Examples (http://micmap.org/php-by-example/en/function/escapeshellarg)
     - `"file.txt"` → `'\'file.txt\''`
     - `"file.txt; ls"` → `'\'file.txt; ls\''`
     - `"file.txt'; ls"` → `'\'file.txt\'\\\'\'; ls\''`

Command Injection: Defenses

3. Use a less powerful and more specific API:
   - Again, the best solution
   - For reading a file, use `file_get_contents()`:
     - Reads the entire file into a string
     - See: http://php.net/manual/en/function.file-get-contents.php
   - For a general OS command, use `proc_open()`:
     - Execute a command and open file pointers for input/output
     - Can only execute *one* command at a time
     - See: http://php.net/manual/en/function.proc-open.php

Additional Reading Materials

SQLI: Techniques

- Tautologies:
  - To make the conditional statements always evaluate to true
  - Sample attack:
    set login to `"' or 1=1 --”`
  - Resulting query:
    `"SELECT accounts FROM users WHERE login=’’ or 1=1 --`
    `AND pass=’’ AND pin=“`
  - Result: bypassing authentication

SQLI: Techniques

- Illegal/Logically Incorrect Queries:
  - To gather important information about the database type/structure
  - Sample attack:
    Set pin to "`convert(int,(select top 1 name from sysobjects where xtype='u'))"`
  - Resulting query:
    `"SELECT accounts FROM users WHERE login='' AND pass='' AND pin=convert(int,(select top 1 name from sysobjects where xtype='u'))"`
  - Output (Microsoft SQL Server):
    `"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."`
  - Results: database finger-printing, obtaining database schema

SQLI: Techniques

- Union Query:
  - To make the application return data from a table different from the one intended by the developer
  - Sample attack:
    set login to `"' UNION SELECT cardNo from CreditCards where acctNo=10032 --"`
  - Resulting query:
    `"SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND pass='' AND pin="`
  - Result: extracting data

SQLI: Techniques

- Piggy-Backed Queries:
  - To inject additional queries into the original one
  - Sample attack (i.e. performing denial of service): set pass to `"'; drop table users --"`
  - Resulting query:
    `"SELECT accounts FROM users WHERE login='doe' AND pass='; drop table users --' AND pin=123"`
  - Result: the injected second query get executed, thus deleting table users

SQLI: Techniques

- Inference:
  - Used in a scenario where the database gives no feedback via database error messages
  - To recast the query into an action that is executed based on the answer to a true/false question about data values in the database
  - Variants: blind SQL injection, timing attacks
- Alternate Encodings: for evading detection
  - Sample attack:
    Set login to `"legalUser'; exec(0x73687574646f776e) --"`
  - Resulting query:
    `"SELECT accounts FROM users WHERE login='legalUser';exec(char(0x73687574646f776e)) -- AND pass='' AND pin="`
  - Result: evade detection and execute a SHUTDOWN external command

SQLI: Some Defenses (PHP)

- `magic_quotes_gpc()`:
  - Used to be on by default, is deprecated now
  - Runs input through `addslashes()`
    - E.g. `')- admin` becomes `\')– admin`
  - Applied to GPC (Get/Post/Cookie) operations
  - Ref: http://php.net/manual/en/info.configuration.php#ini.magic-quotes-gpc
  - Still unsafe:
    - E.g `SELECT * FROM X WHERE id=$post_id`
      - E.g. `0 or 1=1`
    - Native Character set issues

SQLI: Some Defenses (PHP)

- Apply an input escaping using `mysqli_real_escape_string()`
- Ref: http://php.net/manual/en/mysqli.real-escape-string.php
- Possible complication with `magic_quotes_gpc()`:
  - Test magic quote state first, and run `stripslashes()` if needed
  - See sample code in the next slide

SQLI: Some Defenses (PHP)

```php
<?php
// If magic quotes are enabled
echo $_POST['lastname'];              // O\'reilly
echo addslashes($_POST['lastname']); // O\\\'reilly
// Usage across all PHP versions
if (get_magic_quotes_gpc()) {
    $lastname = stripslashes($_POST['lastname']);
}
else {
    $lastname = $_POST['lastname'];
}
// If using MySQL
$lastname = mysqli_real_escape_string($lastname);
echo $lastname; // O\'reilly
$sql = "INSERT INTO lastnames (lastname) VALUES ('$lastname')";
?>
```

Summary

- Server-side injection attacks
  - SQL injection
  - Command injection
- Root cause and other injection attacks?