SQL INJECTION

SQL INJECTION

SQL injection has been one of the **top 10** security vulnerabilities in the world for the last 10 years

In **2013** it was the **number #1** attack as determined by the Open Web Application Security Project

SQL INJECTION

SQL injection is a **code injection technique** used to attack **data driven applications** where **malicious** SQL statements are **inserted** into fields for **execution**

SQL INJECTION

SQL injection is a **code injection technique** used to attack data driven applications where malicious SQL statements are inserted into fields for execution

Just like XSS injects scripts into web pages, SQL injection involves injecting snippets into SQL statements that may be executed on your database

SQL INJECTION

SQL injection is a code injection technique used to attack data driven applications where malicious SQL statements are inserted into fields for execution

Websites are dynamic, they have a database attached to it which holds all kinds of information

SQL INJECTION

SQL injection is a code injection technique used to attack data driven applications where malicious SQL statements are inserted into fields for execution

The information can be sensitive personal information, financial information anything!

SQL INJECTION

SQL statements to query this data are usually constructed with user input

SQL injection is a code injection technique used to attack data driven applications where malicious SQL statements are inserted into fields for execution

SQL INJECTION

The user can inject malicious content to be executed agains the site's database

SQL injection is a code injection technique used to attack data driven applications where **malicious** SQL statements are **inserted** into fields for **execution**

SQL INJECTION

SQL injection is a **code injection technique** used to attack **data driven applications** where **malicious** SQL statements are **inserted** into fields for **execution**

SQL INJECTION

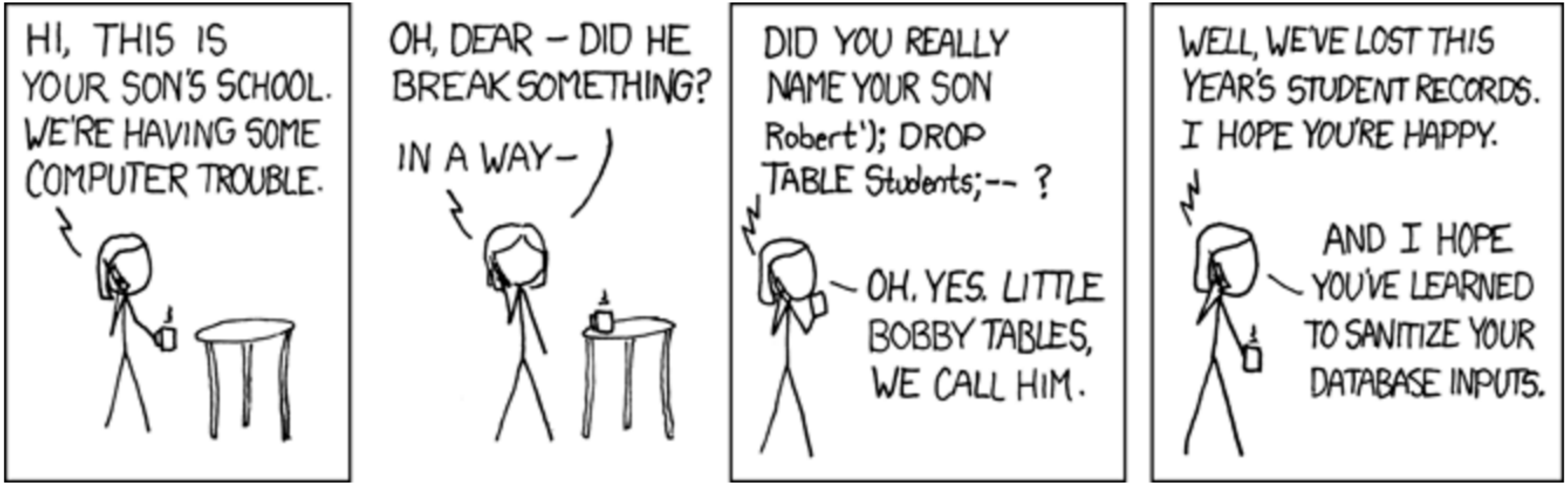
Before we get to understanding how SQL injection works, there is an important thing you should know

WHO IS BOBBY TABLES?

SQL INJECTION

WHO IS BOBBY TABLES?

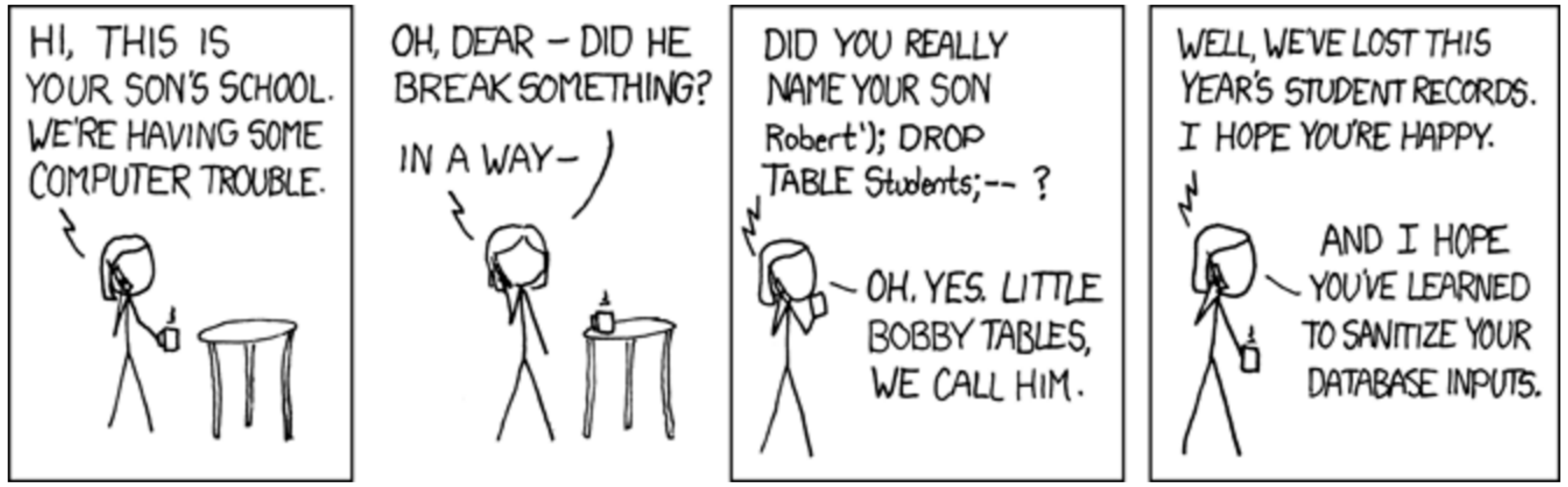
http://xkcd.com/327/



SQL INJECTION

WHO IS BOBBY TABLES?

http://xkcd.com/327/



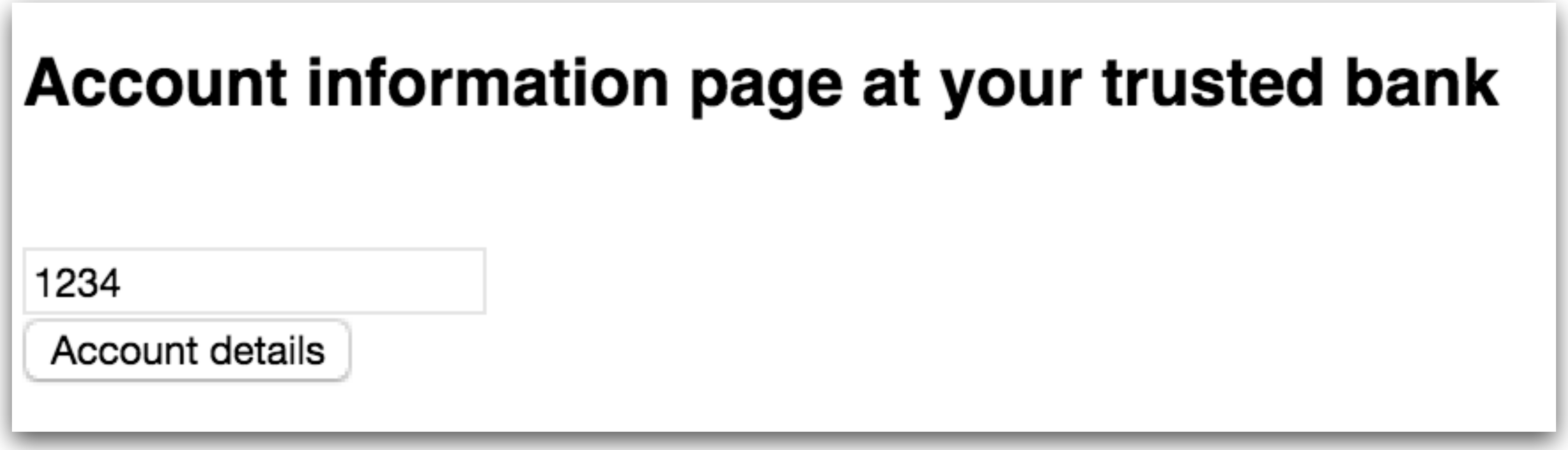
And this in one single, awesome comic is what SQL injection is all about

SQL INJECTION

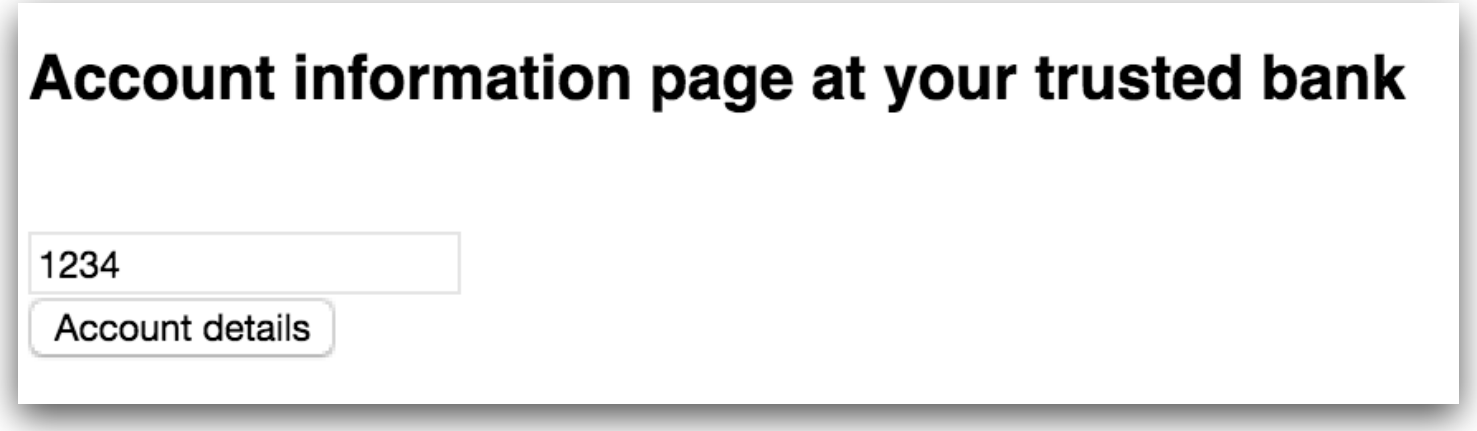
Example13-SQLInjection-simple.php

SQL INJECTION

```
<html>
<body>
<h3> Account information page at your trusted bank</h3>
<form method="GET" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  <br>
  <input type="text" name="account_id" value="1234" readonly>
  <br>
  <input type="submit" value="Account details">
</form>
<br>
</body>
</html>
```



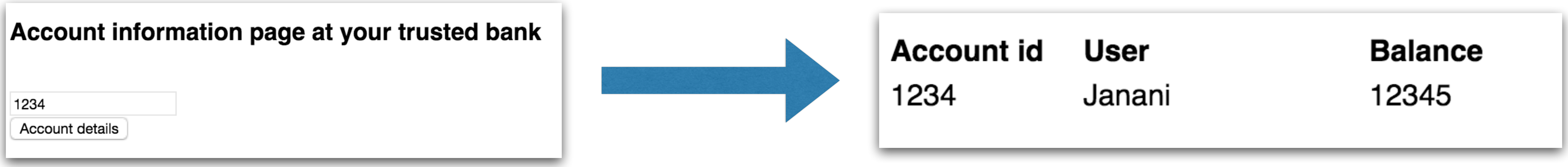
SQL INJECTION



This allows you to view your own account details

The input is not editable and clicking on account details should display your balance

SQL INJECTION



The account id to display is passed as a GET parameter in the url

SQL INJECTION

The account id to display is passed as a GET parameter in the url

http://localhost/security/Example13-SQLInjection-simple.php?account_id=1234

Account id	User	Balance
1234	Janani	12345

SQL INJECTION

http://localhost/security/Example14-SQLInjection-simple.php?account_id=1234

What if you edited the URL to be:

http://localhost/security/Example14-SQLInjection-simple.php?account_id=$\boxed{\texttt{1234 AND TRUE}}$

SQL INJECTION

What if you edited the URL to be:

```
http://localhost/security/Example14-SQLInjection-simple.php?account_id=1234 AND TRUE
```

Account id	User	Balance
1111	Pradeep	100953
1234	Janani	12345
2222	Vitthal	997

We've just accessed all the accounts and balances in the database!

SQL INJECTION

http://localhost/security/Example14-SQLInjection-simple.php?account_id=$\boxed{\text{1234 AND TRUE}}$

Account id	User	Balance
1111	Pradeep	100953
1234	Janani	12345
2222	Vitthal	997

How?

SQL INJECTION

```
$account_id = $_GET['account_id'];

if (!empty($account_id)) {
 try {
    $conn = getDatabaseConnection();

    $result = $conn->query("SELECT * FROM BankAccounts WHERE account_id = " . $account_id);

    if ($result->num_rows > 0) {
        echo '<br><br><table>';
        echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
        echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
        echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
        while($row = $result->fetch_assoc()) {
          echo '<tr>';
          echo '<td style="width: 100px; height: 18px">' . $row['account_id'] . '</td>';
          echo '<td style="width: 150px; height: 18px">' . $row['user_name'] . '</td>';
          echo '<td style="width: 100px; height: 18px">' . $row['account_balance'] . '</td>';
          echo '</tr>';
        }
        echo '</table>';
    } else {
        echo "<br><br>No results match your search:-(";
    }
    mysqli_close($conn);
  } catch (Exception $e) {
    echo 'Error! ' + $e->getCode();
  }
}
```

SQL INJECTION

```
$account_id = $_GET['account_id'];

if (!empty($account_id)) {
 try {
   $conn = getDatabaseConnection();

     $result = $conn->query("SELECT * FROM BankAccounts WHERE account_id = " . $account_id);

   if ($result->num_rows > 0) {
       echo '<br><br><table>';
       echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
       echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
       echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
       while($row = $result->fetch_assoc()) {
         echo '<tr>';
         echo '<td style="width: 100px; height: 18px">' . $row['account_id'] . '</td>';
         echo '<td style="width: 150px; height: 18px">' . $row['user_name'] . '</td>';
         echo '<td style="width: 100px; height: 18px">' . $row['account_balance'] . '</td>';
         echo '</tr>';
       }
       echo '</table>';
   } else {
       echo "<br><br>No results match your search:-(";
   }
   mysqli_close($conn);
  } catch (Exception $e) {
   echo 'Error! ' . $e->getMessage();
  }
}
```

This is the query run to get the details of a particular account

SQL INJECTION

```
$account_id = $_GET['account_id'];

if (!empty($account_id)) {
 try {
    $conn = getDatabaseConnection();

      $result = $conn->query("SELECT * FROM BankAccounts WHERE account_id = " . $account_id);

    if ($result->num_rows > 0) {
        echo '<br><br><table>';
        echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
        echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
        echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
        while($row = $result->fetch_assoc()) {
          echo '<tr>';
          [illegible]
          [illegible]
          [illegible]
          echo '</tr>';
        }
        echo '</table>';
    } else {
        [illegible]
    }
    mysqli_close($conn);
  } catch (Exception $e) {
    echo 'Error! ' + $e->getCode();
  }
}
```

This simply **appends** the account id from the URL parameter to the end of the SQL statement

SQL INJECTION

```
“SELECT *
FROM BackAccounts
WHERE AccountId = “ . $_GET[‘account_id’]
```

The user input is appended to the very end

SQL INJECTION

```
SELECT *
FROM BackAccounts
WHERE AccountId = 1234
```

Which means the user can add in anything he wants in place of the account id

SQL INJECTION

```
SELECT *
FROM BackAccounts
WHERE AccountId = 1234 OR TRUE
```

This selects **all the rows** in the database because the WHERE clause is TRUE

SQL INJECTION

Non-validated string literals are used to **construct** dynamic SQL statements and interpreted as code by the SQL engine

SQL INJECTION

Other examples

SQL INJECTION

```
“SELECT id FROM Users
WHERE username=’” + name + “’ AND password=’” + pass + “’”
```

This is a highly vulnerable SQL statement

SQL INJECTION

```
"SELECT id FROM Users
WHERE username='" + name + "' AND password='" + pass + "'"
```

If our input in the password table looked something like

```
password' OR 1=1 –
```

SQL INJECTION

```
password' OR 1=1 —
```

```
SELECT id FROM Users
WHERE username='someusername'
AND password='password' OR 1=1 —'
```

The user specified password has been literally placed into the SQL statement

SQL INJECTION

```
SELECT id FROM Users
WHERE username='someusername'
AND password='password' OR 1=1 –'
```

This once again matches all the rows in the Users table

SQL INJECTION

```
SELECT id FROM Users
WHERE username='someusername'
AND password='password' OR 1=1 –'
```

The "—" comments out the rest of the SQL statement, so even if the query was more complicated and had additional clauses they are commented out!

SQL INJECTION

```
SELECT id FROM Users
WHERE username='someusername'
AND password='password' OR 1=1 --'
```

In this example the extra **single quote** ' is commented out

SQL INJECTION

The comment character is different for different databases

```
-- MySQL, MSSQL, Oracle, PostgreSQL, SQLite
' OR '1'='1' --
' OR '1'='1' /*
-- MySQL
' OR '1'='1' #
-- Access (using null characters)
' OR '1'='1' %00
' OR '1'='1' %16
```

SQL INJECTION

Getting back to Bobby Tables

If a student were named:

```
Robert'); DROP TABLE Students;--
```

Say this entire name was inserted into a table

SQL INJECTION

Getting back to Bobby Tables

If a student were named:

```
Robert'); DROP TABLE Students;--
```

This SQL would have been executed!

SQL INJECTION

Getting back to Bobby Tables

If a student were named:

```
Robert'); DROP TABLE Students;--
```

Everything else commented out

SQL INJECTION

```
Robert'); DROP TABLE Students;--
```

No wonder the school was unhappy...

SQL INJECTION

Anatomy of an attack

SQL INJECTION

Anatomy of an attack

Thanks to http://www.unixwiz.net/techtips/sql-injection.html for this example

SQL INJECTION

Anatomy of an attack

So how does an attacker go about a SQL injection attack?

SQL INJECTION

Anatomy of an attack

The objective is to log into a website using a valid username and password

SQL INJECTION

Anatomy of an attack

Consider an example where a website has a "**email me my password**" feature

in case the user has forgotten his password

SQL INJECTION

Anatomy of an attack

Consider an example where a website has a **"email me my password"** feature

The user enters an email address and **if it is found**

The password associated with that email is mailed to that address

SQL INJECTION

Anatomy of an attack

Consider an example where a website has a **email me my password** feature if the user has forgotten his password

The user enters an email address and **if it is found**

The password associated with that email is mailed to that address

Seems fairly straightforward...

SQL INJECTION

Anatomy of an attack

An attacker preparing an attack will first check to see whether the **input data is being sanitized or not**

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = '<user input email>'
```

Assume this is what the basic structure of the query looks like

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = '<user input email>'
```

The <user input email> comes from the user input in a form

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = '<user input email>'
```

We enter:

```
jan@loonycorn.com'
```

SQL INJECTION

Anatomy of an attack

input data sanitized or not

```
SELECT * FROM Users
WHERE email = '<user input email>'
```

```
jan@loonycorn.com'
```

This addition of a quote serves to check whether the data is used **literally** or not

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = ‘jan@loonycorn.com’’
```

If the server runs this statement this will be a SQL error

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = ‘jan@loonycorn.com’’
```

If the server runs this statement this will be a SQL error

How the error is handled by the server is information for the attacker

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'jan@loonycorn.com''
```

If a nice message is returned to the user it means the server sanitized the input or handled the error

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'jan@loonycorn.com''
```

If the site returns an error - then it means that the input was used literally without sanitization

input data sanitized or not

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'jan@loonycorn.com''
```

The attacker will probably see "Internal error" or "Database error" or an exception stack trace

SQL INJECTION

Anatomy of an attack

It seems like the email comparison is in the WHERE clause

We enter:

```
some email' or '1'='1
```

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' or '1'='1'
```

'1' = '1' will always be true - the nature of the clause has been changed in an entirely legal way

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' or '1'='1'
```

From a single component clause this becomes a 2 component clause where '1' = '1' is **always** true!

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' or '1'='1'
```

So what happens now?

There will be at least one match for this query - let's say an **email is sent to the first match**

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' or '1'='1'
```

The attacker gets a message

"Password has been mailed to someemail@email.com"

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = ‘some email’ or ‘1’=‘1’
```

Some user has possibly received his password which should make him suspicious

but people often ignore such emails

SQL INJECTION

Anatomy of an attack

So far we know:

1. Unsanitized inputs leading to SQL errors give some kind of **server** error

2. **Valid** inputs give **no error** and possible a nice message on screen

Now we want to figure out what the **column names** are in this table

SQL INJECTION

Anatomy of an attack

Use the email input field as before and the fact that correct SQL queries do not result in a server error

```
SELECT * FROM Users
WHERE email = 'some email' AND email is NULL —'
```

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' AND email is NULL --'
```

We don't care about matching the email

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' AND email is NULL –'
```

We want to check whether "**email**" is a **valid column name** in this table

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' AND email is NULL --'
```

If email is a **valid** column name then this is a valid query and there should be **no error** from the server

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' AND email is NULL --'
```

This comments out the rest of the query, whatever it was

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email' AND username is NULL –'
```

Try different column names till you find the ones which cause no errors

SQL INJECTION

Anatomy of an attack

Collect a list of valid column names

email
user_id
password
name

SQL INJECTION

Anatomy of an attack

Now to find the **table name**

The query changes but the **principle** remains the same

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email'
AND 1=(SELECT COUNT(*) FROM table_name); --'
```

If the table exists then there will be no error

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email'
AND 1=(SELECT COUNT(*) FROM table_name); --'
```

If we find a valid table there is no guarantee that this is the table which stores the email information - it is **simply a table** in the database

SQL INJECTION

Anatomy of an attack

Let's say we find the table name

Users

```
SELECT * FROM Users
WHERE email = 'some email'
AND AND Users.email IS NULL;; --'
```

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email'
AND AND Users.email IS NULL;; --'
```

This will be error free only if there is a **match** between the **table queried** and the **table we guessed**

SQL INJECTION

Anatomy of an attack

So far we know:

1. Valid column names for the table queried

2. Valid table name

SQL INJECTION

Anatomy of an attack

Now we want to figure out any users that might exist in this table

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = 'some email'
OR name LIKE '%bob%'
```

We can try a whole bunch of common names and email extensions such as gmail.com, yahoo.com etc

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = ‘some email’
OR name LIKE ‘%bob%’
```

If we get a match we’ll get a message of the type “your password has been mailed to someemail@email.com”

SQL INJECTION

Anatomy of an attack

```
SELECT * FROM Users
WHERE email = ‘some email’
OR name LIKE ‘%bob%’
```

We now have a valid user email from this website!

SQL INJECTION

Anatomy of an attack

At this point we can cause a whole lot of havoc in this database

Delete entire tables if the database is not readonly

SQL INJECTION

Anatomy of an attack

Delete entire tables if the database is not readonly

```
SELECT * FROM Users
WHERE email = 'some email';
DROP TABLE Users; --'
```

SQL INJECTION

Anatomy of an attack

There is a tricky way the blog used to get access to the username + password

SQL INJECTION

Anatomy of an attack

1. Updated the email address to the email of the attacker i.e his own email

2. Mailed himself the password by clicking on lost password

SQL INJECTION

Anatomy of an attack

```
SELECT *
FROM Users
WHERE email = ‘some email';
        UPDATE Users
        SET email = ‘jan@loonycorn.com’
        WHERE email = 'bob@example.com';
```

Now the attacker’s email is present in the web site’s database!

SQL INJECTION

Anatomy of an attack

```
SELECT *
FROM Users
WHERE email = ‘some email';
        UPDATE Users
        SET email = ‘jan@loonycorn.com’
        WHERE email = 'bob@example.com';
```

Now just enter this email into the "email me my password" input box and receive email with the password

SQL INJECTION

Anatomy of an attack

```
SELECT *
FROM Users
WHERE email = ‘some email';
        UPDATE Users
        SET email = ‘jan@loonycorn.com’
        WHERE email = 'bob@example.com';
```

We have a valid login to the site!

SQL INJECTION

Anatomy of an attack

This is a classic example of a **blind SQL injection** attack!

SQL INJECTION

Anatomy of an attack

This is a classic example of a **blind SQL injection** attack!

The attacker **cannot see the result of the attack** but can use the response of the server to make guesses

SQL INJECTION

Anatomy of an attack

This is a classic example of a **blind SQL injection** attack!

This involves a lot **patient maneuvering** and **crafting** of SQL statements for every piece of data recovered

SQL INJECTION

Anatomy of an attack

Second order SQL injection occurs when **submitted** values contain malicious commands which are stored in the site's database

SQL INJECTION

Anatomy of an attack

Second order SQL injection occurs when **submitted** values contain malicious commands which are stored in the site's database

Another part of the site which does **not have injection controls** might execute those commands and expose data

SQL INJECTION

Types of SQL Injection

SQL INJECTION

Types of SQL Injection

1. In-band SQLi

2. Blind SQLi

3. out-of-band SQLi

SQL INJECTION

In-band SQLi

This is a kind of SQL attack when an attacker is able to use the same **communication channel** to both

launch the attack

gather the results

SQL INJECTION

In-band SQLi

As in the anatomy of an attack the **form input field** is used to launch the attack and **the results of specifying the input** is used to gain information

SQL INJECTION

In-band SQLi

This is the most **common** and easy to launch attack

SQL INJECTION

In-band SQLi

This can be of two types:

Error based SQLi

Union based SQLi

Union based SQLi

SQL INJECTION

In-band SQLi

Error based SQLi

This relies on **error messages** thrown by the database server to obtain information about its structure

Union based SQLi

SQL INJECTION

In-band SQLi

Error based SQLi

The **type** of error, or even the very **existence** of the error is information to the attacker

Union based SQLi

SQL INJECTION

In-band SQLi

Error based SQLi

While errors are great during development, the **production** environment should not display errors to the user - they can be **logged to a restricted file** instead

Error based SQLi

SQL INJECTION

In-band SQLi

Union based SQLi

This involves use of a **UNION** to include **another SELECT** statement with the original SQL command

Error based SQLi

SQL INJECTION

In-band SQLi

Union based SQLi

The response will contain the results of the original statement as well as the **results of the new statement** which was **injected**

Error based SQLi

SQL INJECTION

In-band SQLi

Union based SQLi

```
SELECT name, email
FROM Users
WHERE email = ‘some email';
          UNION
SELECT name, email
FROM Users;
—‘;
```

If you know the original format of the statement you can just append your own!

SQL INJECTION

Types of SQL Injection

1. In-band SQLi
   - Error based SQLi
   - Union based SQLi
2. Blind SQLi
3. out-of-band SQLi

SQL INJECTION

Blind SQLi

In such an attack **no data is transferred** along with the web application

the attacker may not necessarily see the result of his attack

SQL INJECTION

Blind SQLi

The attacker tries to **reconstruct** the database structure by sending payloads, **observing** the server response and the resulting **behavior**

SQL INJECTION

Blind SQLi

This might take much longer to figure out an exploit, based on **trial and error**

SQL INJECTION

Blind SQLi

This can be of two types:

Boolean based SQLi

Time based SQLi

Time based SQLi

SQL INJECTION

Blind SQLi

Boolean based SQLi

This relies on sending a SQL query which forces the server to return a **different** result if the query evaluates to **TRUE** or **FALSE**

Time based SQLi

SQL INJECTION

Blind SQLi

Boolean based SQLi

The server response **changes** based on whether the query was TRUE or FALSE allowing the attacker to infer the payload response

Boolean based SQLi

SQL INJECTION

Blind SQLi

Time based SQLi

This involves sending a query to the server which forces **the database to wait** a specified amount of time before responding

Boolean based SQLi

SQL INJECTION

Blind SQLi

Time based SQLi

The attacker draws inferences based on whether the server response was **immediate or delayed**

SQL INJECTION

Types of SQL Injection

1. In-band SQLi

Error based SQLi

Union based SQLi

2. Blind SQLi

Boolean based SQLi

Time based SQLi

3. out-of-band SQLi

SQL INJECTION

Out-of-band SQLi

This is not a commonly used attack as it relies on the database server having certain features enabled

SQL INJECTION

Out-of-band SQLi

Perhaps the database server's can make **DNS** or **HTTP** requests to deliver data to an attacker

SQL INJECTION

Out-of-band SQLi

An example is Microsoft SQL server's **xp_dirtree** command which can be used to make DNS requests to the attacker's server

SQL INJECTION

Types of SQL Injection

1. In-band SQLi
   - Error based SQLi
   - Union based SQLi

2. Blind SQLi
   - Boolean based SQLi
   - Time based SQLi

3. out-of-band SQLi

SQL INJECTION

Mitigation

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation

SQL INJECTION

Parameterized statements

The use of **prepared** statements with **parameters** is the **safe** and correct way to write SQL queries

SQL INJECTION

Parameterized statements

The use of **prepared** statements with **parameters** is the **safe** and correct way to write SQL queries

These are simpler and easier to write than dynamic queries

SQL INJECTION

Parameterized statements

The use of **prepared** statements with **parameters** is the **safe** and correct way to write SQL queries

This forces the developer to **think through** what the entire SQL query structure looks like and what parts are **filled in with user data**

SQL INJECTION

Parameterized statements

The use of **prepared** statements with **parameters** is the **safe** and correct way to write SQL queries

It **differentiates** clearly between **code** (the actual query) and **data** (user input)

SQL INJECTION

Example14-SQLInjection-parameterizedQueries.php

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);

echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);
echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

Set up the query to make to the database

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);
echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

The ? is a placeholder for the user input account id

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);

echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

Bind the user specified account id to the prepared statement

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);
echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

The “i” indicates that the account id should be interpreted as an integer

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);
echo '<br><br><table>';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

Execute the prepared statement

SQL INJECTION

Parameterized statements

```
$conn = getDatabaseConnection();

$stmt = $conn->prepare("SELECT * FROM BankAccounts WHERE account_id = ?");
$stmt->bind_param("i", $account_id);
$stmt->execute();

$stmt->bind_result($account_id, $user_name, $account_balance);
echo '[illegible]';
echo '<td style="width: 100px; height: 22px">' . "<b>Account id</b>" . '</td>';
echo '<td style="width: 150px; height: 22px">' . "<b>User</b>" . '</td>';
echo '<td style="width: 100px; height: 22px">' . "<b>Balance</b>" . '</td>';
while($stmt->fetch()) {
  echo '<tr>';
  echo '<td style="width: 100px; height: 18px">' . $account_id . '</td>';
  echo '<td style="width: 150px; height: 18px">' . $user_name . '</td>';
  echo '<td style="width: 100px; height: 18px">' . $account_balance . '</td>';
  echo '</tr>';
}
echo '</table>';
```

And bind the result to variables

SQL INJECTION

Parameterized statements

Prepared statements allow the user to specify the intent and the **attacker cannot change** this intent

SQL INJECTION

Parameterized statements

```
SELECT id FROM Users
WHERE username= ?
AND password= ?
```

Let's say input to username is "jan OR TRUE"

SQL INJECTION

Parameterized statements

```
SELECT id FROM Users
WHERE username='jan OR TRUE'
AND password= ?
```

And the input in the password field is "password' OR 1=1 --"

SQL INJECTION

Parameterized statements

```
SELECT id FROM Users
WHERE username='jan OR TRUE'
AND password='password OR 1=1 –'
```

The statement will actually look for a user name and password which **literally** matches the string passed in

SQL INJECTION

Parameterized statements

In some cases prepared statements can **harm performance** - in such specific circumstances we can resort to other techniques to protect against SQL injection

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation

SQL INJECTION

Stored procedures

A stored procedure is a bunch of SQL statements which are **logically grouped** to perform a specific task

SQL INJECTION
Stored procedures

The statements are compiled and work **kind of as a function** does in a programming language

Stored procedures

Typically stored procedures work like parameterized statements - unless the developer goes out of the way to do something different

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation

SQL INJECTION

Escaping user input

This involves escaping any characters which have a **special** meaning in SQL

SQL INJECTION

Escaping user input

Typically every database has a list of characters which should be escaped

There usually is a **method** which is the right one to use to escape user input

SQL INJECTION

Escaping user input

```
$mysqli = new mysqli('hostname', 'db_username', 'db_password', 'db_name');
$query = sprintf("SELECT * FROM `Users` WHERE UserName='%s' AND Password='%s'",
                  $mysqli->real_escape_string($username),
                  $mysqli->real_escape_string($password));
$mysqli->query($query);
```

If you use PHP and MySQL the **real_escape_string**() method on the connection escapes user input for MySQL

SQL INJECTION

Escaping user input

```
$mysqli = new mysqli('hostname', 'db_username', 'db_password', 'db_name');
$query = sprintf("SELECT * FROM `Users` WHERE UserName='%s' AND Password='%s'",
                  $mysqli->real_escape_string($username),
                  $mysqli->real_escape_string($password));
$mysqli->query($query);
```

PHP has other escaping functions for different database types such as **pg_escape_string**() for PostgreSQL

SQL INJECTION
Escaping user input

Another way of validating input is to check whether the input is a **valid representation of the type** i.e. integer for integer columns and so on

SQL INJECTION

Escaping user input

Instead of placing the burden on every developer to escape inputs, typically a **layer between the database and application** should take care of this

Escaping user input

One particular variation of escaping input is to **hex-encode** all input data

SQL INJECTION

Escaping user input

One particular variation of escaping input is to **hex-encode** all input data

This means only characters **0-9** and **a-f** are valid in user input, SQL special characters also get encoded to this format

SQL INJECTION

Escaping user input

One particular variation of escaping input is to **hex-encode** all input data

This means only characters **0-9** and **a-f** are valid in user input, SQL special characters also get encoded to this format

The **comparison** with values in the database should account for the hex-encoded form

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation

SQL INJECTION

Least privilege

Every database account in the database should only be given **sufficient privilege** to perform the tasks required

The **least** privilege possible

SQL INJECTION

Least privilege

Do not give any application level accounts admin privileges

Determine whether accounts need **read** or **write** privileges

Determine **which tables** each account requires access to

SQL INJECTION

Least privilege

Start from the **ground up** to accord privileges to accounts

If an account needs access to only some data from a table use **views** instead

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation

SQL INJECTION

Whitelist validation

Whitelist validation involves figuring out **what exactly is valid** and only allowing inputs which adheres to that

SQL INJECTION

Whitelist validation

For **structured** data like dates, social security numbers, email addresses etc a very **strong** validation pattern can be defined

SQL INJECTION
Whitelist validation

Of these **free text** is the **hardest** to validate, even this can have minimal validation like having a **max length** defined, allowing only **printable characters** etc.

SQL INJECTION

Mitigation

1. Parameterized statements
2. Stored procedures
3. Escaping user input
4. Least privilege
5. Whitelist validation