SQL Injection

White Hats: Team Members
Banks Cargill - cargillb@oregonstate.edu
Frederick Eley - eleyf@oregonstate.edu
Timothy Glew - glewt@oregonstate.edu

Description:

The first web application security risk listed by the OWASP Foundation is injection. Injection can occur when untrusted data is sent to an interpreter as part of a command or query. Malicious data sent by an attacker can execute commands not intended by the programmer or grant unauthorized access to stored data.

One of the most common interpreted languages used for web application databases is SQL and it is what we will focus our attention on for this risk. SQL injections are typically performed using web page forms in which a user is allowed to provide input. Examples of such are form fields and search boxes. This input then becomes part of a command or query to the SQL database. All SQL verbs such as SELECT, INSERT, UPDATE, and DELETE can all be exploited if injection occurs.

A web application is vulnerable to SQL injection when user-supplied data is not validated, filtered, or sanitized. This is particularly troublesome when that data is directly used or concatenated in the construction of the SQL query. This vulnerability can be found mainly in older code as modern APIs are built to safeguard against SQL injection attacks. That does not exclude modern applications from this risk as the improper implementation of such APIs will allow injection to occur.

SQL injection attacks are categorized into two main types, error-based SQL injection and blind SQL injection. In error-based attacks, bad input is injected with the intent to cause the database to throw an error. An attacker is then able to read the database-level error message and create a customized injection to compromise the application. An example of an error-based attack is the tautology-based attack. In a tautology-based attack, the code is injected using the conditional 'OR' operator in the 'WHERE' clause of a SQL query such that the query always evaluates to TRUE. Tautology-based SQL injection attacks are used as a method to bypass user authentication and extract data. In blind-based attacks there are no error messages from the application which can leak information about the internal workings. One way an attacker can still gather information by injecting both TRUE and FALSE statements and comparing the response pages.

How to Attack our Site:

Tautological Injection - Login

At the login page, you'll need to enter text into the username and password to bypass the form requirement. At the end of the password entry add: ' OR user_id like '%%

This will make the query read as:

```
SELECT * FROM users WHERE username = 'yourinput' AND pword =
'yourinput' OR user_id like '%%'
```

This will return all rows from the 'users' table with a user_id. Because the front end of the web application assigns the user information based on the first row, you will be logged in as if you were the first returned user_id.

This attack is based on the assumption that the attacker can guess a name for the property that holds the user's id. Since most databases we have seen use "user_id" or "id", we felt this was a safe assumption to make.

Once you have logged in and proven both that user_id is a property of the user table and the injection works, you can now play with different user_id's and log in as different users.

For example:

```
'OR user_id = '2
```

SQL Injection - User Input Forms (Add List Example)

Without proper defense mechanisms in place, our user input forms to add tasks and lists are vulnerable to SQL injection. Starting on the homepage where your 'To-Do Lists' are shown we can begin by adding a new list which will execute the following query:

```
INSERT INTO `lists` (name, description) VALUES ('[user_input]',
'[user_input]')
```

We can use the 'Description' box to inject arbitrary queries. For example, you could enter the following to delete the tasks table from the database (replace 'any_name' and 'any_desc' with the name and description for a list):

```
List Name: any_name   Description: any_desc');SET FOREIGN_KEY_CHECKS=0;
DROP TABLE tasks;#
```

This will run the following query:

```
INSERT INTO `lists` (name, description) VALUES (‘any_name’,
‘any_desc');SET FOREIGN_KEY_CHECKS=0; DROP TABLE tasks;#’)
```

Appening **‘);** to the end of the description entered will end the query to add your list, which is followed by two queries that together will drop the tasks table. The **#** is a comment character that indicates the rest of the input is a comment and should not be parsed as part of the query statement, so the final **‘)** will be ignored. Entering the above information into the ‘Add a New List’ form will take you to an error page, so renavigate to the home page with the following URL:

```
osu-capstone-project-insecure.herokuapp.com/home
```

You will see that your list was added, but you are now unable to view the tasks for any of the lists since the table has been deleted. Clicking on any of the ‘View’ buttons will instead take you to an error page. If you execute the above injection you can reset the database to return to normal. The same vulnerability in adding a list shown above is also present in the form for adding a task to a list, so the same types of injection can be done through that form as well.

Blind Injection - View Tasks

We can also apply blind SQL injection techniques to determine if other parts of our site are susceptible to SQL injection. Once logged in, if you don’t already have any existing to-do list, add one. Then click ‘view’ to see all the tasks on the list (lists are empty when created, so feel free to add a few tasks if you are viewing a new list). When you click ‘view’ you will be taken to the following URL where the ‘2’ will be the id of the list you selected to view:

```
osu-capstone-project-insecure.herokuapp.com/tasks/2
```

The above example will execute the following query to determine which tasks to show you:

```
SELECT task_desc, completed, type FROM `tasks` WHERE list_id=2
```

To begin testing if SQL injection through the URL is possible, you can start by attempting to inject a query that will return ‘false’ and seeing what the webpage returns by entering the following URL:

```
osu-capstone-project-insecure.herokuapp.com/tasks/2 AND 1=2
```

Entering this URL will execute the following query, which will always be false since 1 ≠ 2:

```
SELECT task_desc, completed, type FROM `tasks` WHERE list_id=2 AND 1=2
```

We do not expect this to return the tasks associated with the current list since the 'WHERE' clause in the query should evaluate to false. The next thing to do is verify that we did not have our tasks returned because of our injected query. We can do this by injecting another query that should return true, and verifying that our tasks are appropriately displayed. To do this, you can inject the following query, which will always be true since 1 = 1:

```
osu-capstone-project-insecure.herokuapp.com/tasks/2 AND 1=1
```

The above URL returns our tasks like normal which confirms that our injected query was run and the tasks portion of our site is vulnerable to SQL injection through the URL. One way to use this to attack our site is to use the following URL to view all tasks in the database:

```
osu-capstone-project-insecure.herokuapp.com/tasks/2 OR 1=1
```

Entering the URL above will run the following query:

```
SELECT task_desc, completed, type FROM `tasks` WHERE list_id=2 OR 1=1
```

This query works because we are asking for all tasks where the list_id is 2 OR where 1=1, which is always true, so all tasks in the database, whether they are associated with our list or not, will satisfy this criteria, and thus be displayed.

How to Defend our Site:

Tautological Injection - Login

For our vulnerable site, we wrote the query so that if a user knew a valid username and a valid password that would return a single row from the database, they could login. We changed this so that the database was queried solely from the username input. Since usernames must be unique, this will only return one row from the DB. We then check the user's input for their password against the DB's returned password. If they match, the user is logged in.

To further protect against users injecting queries, we created a stored procedure. A stored procedure is a type of parameterised query. A stored procedure writes the query beforehand and earmarks the locations where data will later be supplied. This clearly differentiates what is part of the query, and what is data to be used in the query, that the database understands. This prevents injected queries from being run because any query that someone attempts to inject will be interpreted as data rather than as a query.

Insecure: Dynamic Query Construction

```
query = "SELECT * FROM users WHERE `username`='{}' AND pword='{}'".format(username, password)
```

Secure: Stored Procedure

```
CREATE DEFINER=`a4dp6xjzj6ogrgmu`@`%` PROCEDURE `returnUserInfo`(IN uname varchar(20))
BEGIN
    SELECT * FROM users WHERE username = uname;
END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('returnUserInfo', [username, ])
result = cursor.fetchall()
```

Secure: Cross Validation of User Input Password and DB-stored Password

```
if result:
    #added this as validation that user input matched query results
    if username == result[0][1] and password == result[0][2]:
        user = User(user_id=result[0][0], username=result[0][1], password=result[0][2], email=result[0][3])
        login_user(user)
        flash('You have been logged in!', 'success')
        next_page = request.args.get('next')
        db_connection.close() # close connection before returning
        return redirect(url_for('home'))
```

SQL Injection - User Input Forms

As discussed in the previous section about our defense against injection on the login page, to prevent SQL injection into our input forms we created stored procedures within our database to replace the dynamically built queries previously used.

<u>Adding a List:</u>

Insecure: Dynamic Query Construction

```
query = "INSERT INTO `lists` (`user_id`, `name`, `description`) VALUES \
        ('{}', '{}', '{}')".format(inputs['user_id'], inputs['list_name'], inputs['list_desc'])
execute_query(db_connection, query) # execute query
```

Secure: Stored Procedure

```
CREATE DEFINER=`user_name`@`%` PROCEDURE `addList`(IN user_id int(11), name varchar(20), description varchar(80))
BEGIN
    INSERT INTO lists (user_id, name, description) VALUES (user_id, name, description);
END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('addList', [inputs['user_id'], inputs['list_name'], inputs ['list_desc'], ])
db_connection.commit()
```

Adding a Task:

Insecure: Dynamic Query Construction

```
query = "INSERT INTO `tasks` (`list_id`, `dataType_id`, `description`, `completed`) \
        VALUES ('{}', '{}', '{}', '{}')".format(inputs['list_id'], inputs['task_type'],
        inputs['task_desc'], inputs['task_comp'])
execute_query(db_connection, query).fetchall()  # execute query
```

Secure: Stored Procedure

```
CREATE DEFINER=`user_name`@`%`
PROCEDURE `addTask`(IN list_id int(11), task_type int(11), description varchar(80), task_comp boolean)
BEGIN
    INSERT INTO tasks (list_id, dataType_id, description, completed) VALUES (list_id, task_type, description, task_comp);
END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('addTask', [inputs['list_id'], inputs['task_type'], inputs['task_desc'], inputs['task_comp'], ])
db_connection.commit()
```

Registering a User:

Insecure: Dynamic Query Construction

```
query = ('INSERT INTO `users` '
        '(`user_id`, `username`, `pword`, `email`) '
        'VALUES (NULL, %s, %s, %s);')
data = (username, password, email)
cursor = execute_query(db_connection, query, data)
cursor.close()
```

Secure: Stored Procedure

```
CREATE DEFINER=`a4dp6xjzj6ogrgmu`@`%` PROCEDURE `addUser`(IN username varchar(20),
                                                          pword varchar(20), email varchar(50))
BEGIN
    INSERT INTO users (username, pword, email) VALUES (username, pword, email);
END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('addUser', [username, password, email, ])
db_connection.commit()
cursor.close()
```

Blind Injection - View Tasks

As with the previous two sections, we can prevent blind SQL injection probing and subsequent SQL injection through our view tasks URL by using a stored procedure in place of our dynamically built query.

Insecure: Dynamic Query Construction

```
query = "SELECT tasks.task_id, tasks.list_id, tasks.dataType_id, \
    tasks.description, tasks.completed, dataTypes.name FROM `tasks` \
    JOIN `dataTypes` ON tasks.dataType_id = dataTypes.dataType_id \
    WHERE list_id = '{}'".format(list_id)  # get info of tasks on list
rtn = execute_query(db_connection, query).fetchall()  # run query
```

Secure: Stored Procedure

```
1  CREATE DEFINER=`a4dp6xjzj6ogrgmu`@`%` PROCEDURE `returnTasks`(IN listId int(11))
2  BEGIN
3      SELECT tasks.task_id, tasks.list_id, tasks.dataType_id, tasks.description, tasks.completed, dataTypes.name
4      FROM `tasks` JOIN `dataTypes` ON tasks.dataType_id = dataTypes.dataType_id WHERE tasks.list_id = listId;
5  END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('returnTasks', [list_id, ])
rtn = cursor.fetchall()
context['rows'] = rtn  # rtn = tasks data
```