HyperionDev

SQL Injection

Task

Visit our website

Introduction

In this task, we will introduce a new type of cyber attack, specifically SQL injection(SQLi). This attack can manipulate user input to negatively affect a database. With the right input to an unprotected field, the attacker can delete an entire database, or pull valuable information from it.

Understanding SQL injection in SQLite

When working with SQLite in Python, handling user input securely is crucial to protect your database from SQL injection attacks. SQL injection is a common vulnerability where attackers can manipulate input fields to execute malicious SQL code, potentially gaining unauthorised access to your database.

Consider a scenario where you need to insert user-provided data into a database. Suppose you have the following Python code:

```
cursor = db.cursor()
student_name = 'Andres'
student_grade = 60
cursor.execute('''
               INSERT INTO student(name, grade)
               VALUES(?,?)''',
               (student_name, student_grade))
```

In this example, the `INSERT INTO` statement adds a new student's name and grade to the database. Notice how the SQL command uses placeholders (`?`) for the values, which are then supplied as a tuple in the `execute` method. This method, known as a **prepared statement**, is a secure way of interacting with your database. The SQL command is first compiled into executable code, and then the user-provided values are safely inserted. This separation ensures that the input is treated purely as data, not as part of the SQL code itself, thereby safeguarding your application against SQL injection.

Now, let's consider a less secure approach:

```
cursor.execute(f'''
               INSERT INTO student(name, grade)
               VALUES("{student_name}",{student_grade})''')
```

In this version, the SQL command is constructed using Python's f-string formatting, directly embedding the user input into the SQL statement. While this might seem convenient, it opens the door to potential SQL injection attacks. Since the user input is incorporated into the SQL code before it is compiled, an attacker could inject malicious

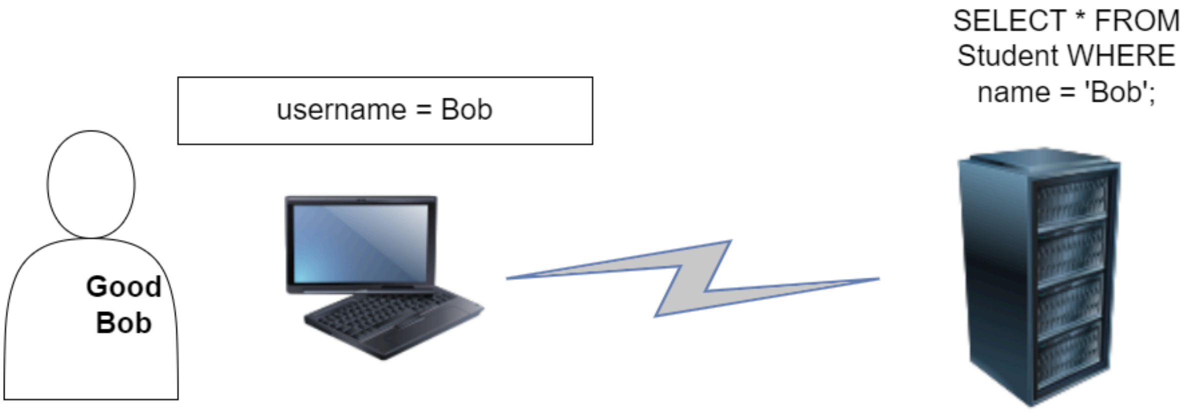
SQL code through the input fields. For instance, if an attacker managed to input a name like `"; DROP TABLE student; --`, it could lead to the deletion of the entire student table, resulting in severe data loss.

Deleting data using SQL injection

We've established that SQL injection involves user input being transformed into runnable code. But how exactly can an attacker exploit this? Let's explore this concept using a simple SQL statement:

```
cursor.execute(f"SELECT * FROM Student WHERE Name = '{student_name}';")
```

In this example, the user is expected to enter their name, and the code executes a SQL `SELECT` statement to retrieve their information from the database. Under normal circumstances, this seems harmless and functions with the best of intentions. For instance, when a well-meaning user, like Good Bob, logs into the system and enters his name, the SQL query would look like this:

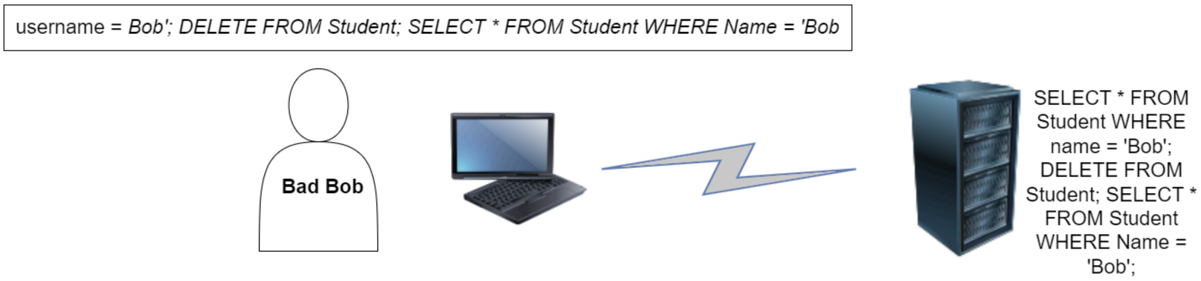


Good Bob logs into the system and retrieves data

However, Good Bob has a twin brother, Bad Bob. Unlike his brother, Bad Bob harbours ill intentions toward the university's system. Determined to cause havoc, he devises a plan to delete all records in the database. After watching some YouTube tutorials and enrolling in the HyperionDev bootcamp, he acquires a few hacking skills. He then logs into the university system and, instead of entering his name, inputs the following into the username textbox:

```
Bob'; DELETE FROM Student; SELECT * FROM Student WHERE Name = 'Bob
```

Let's see what statement is sent to the database:


username = Bob'; DELETE FROM Student; SELECT * FROM Student WHERE Name = 'Bob
Bad Bob
SELECT * FROM Student WHERE name = 'Bob'; DELETE FROM Student; SELECT * FROM Student WHERE Name = 'Bob';

Bad Bob attempts a SQL injection that could delete data from the student database

This query is a textbook example of SQL injection. By cleverly inserting his input, Bad Bob has transformed the intended SQL query into a dangerous sequence of commands that not only tries to retrieve data but also deletes all records from the `Student` table. This demonstrates how seemingly harmless user input can be exploited to execute malicious SQL code, leading to severe consequences if the system is not properly secured.

However, for Bad Bob's attack to succeed, he must have specific knowledge about the system. The effectiveness of this attack depends on the attacker knowing exactly how the SQL statement is structured. Specifically, Bad Bob would need to know that:

- the SQL statement uses single quotes to enclose string values, and
- the name of the table being targeted, which in this case is `Student`.

Accessing user data via SQL injection

Let's look at the following login code:

```
cursor.execute(f'''SELECT * FROM Student
WHERE username = '{user_name}'
AND password = md5('{password}');''')
```

This will only fetch user records if your username is matched in the database and if your password matches as well. Notice the use of **MD5**, which is a hashing function applied to the password. Instead of storing the actual password, the database stores its hash, and the input password is hashed as well for comparison.

Now, imagine you're an attacker targeting the username **me@imaginary.com**. If you've discovered that the server is vulnerable to SQL injection, you could exploit this by entering the following into the username field:

```
me@imaginary.com';--
```

This input works because `--` is the SQL notation for comments, causing everything that follows to be ignored. As a result, the executed SQL statement looks like this:

```
SELECT * FROM Student WHERE username = 'me@imaginary.com';--' AND password =
md5('{password}');
```

This effectively bypasses the password check as the condition `AND password = md5('{password}')` is ignored.

Alternatively, an attacker might try to bypass the password verification entirely. Given that the `md5()` function is used to hash passwords – a common practice for secure password storage – the attacker could input the email address in the username field and the following in the password field:

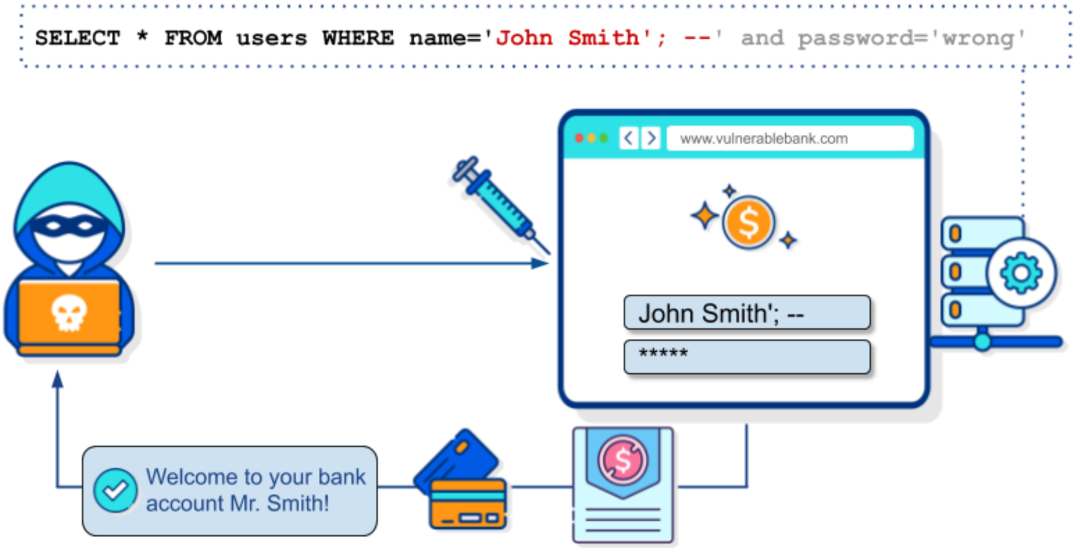
```
') OR TRUE;--
```

This would generate a SQL statement that looks like this:

```
SELECT * FROM Student WHERE username = 'me@imaginary.com' AND password =
md5('') OR TRUE;--');
```

Here, the SQL query attempts to check the `md5` hash of an empty string (`''`), which would typically fail. However, the condition `OR TRUE` creates a situation where the overall expression becomes `False OR True`, which evaluates to `True`. This logic allows the attacker to bypass the password check entirely, gaining unauthorised access to the system.

This example underscores the risks associated with insecure SQL queries. The following visual demonstrates how an attacker can exploit vulnerable SQL queries by injecting malicious code into input fields. In this example, the attacker bypasses authentication by manipulating the username input, tricking the system into granting unauthorised access. This highlights the critical importance of validating user inputs and using secure coding practices to prevent such attacks.


SELECT * FROM users WHERE name='John Smith'; --' and password='wrong'
www.vulnerablebank.com
John Smith'; --
*****
Welcome to your bank account Mr. Smith!

SQL injection (SecureFlag, n.d.)

Preventing SQL injections

Now that we've explored some basic types of SQL injection, the question arises: how can we protect our systems from these attacks? Earlier, we discussed a secure way to execute SQL statements using prepared statements. Let's revisit that example:

```
cursor = db.cursor()
student_name = 'Andres'
student_grade = 60
cursor.execute('''
               INSERT INTO student(name, grade)
               VALUES(?,?)''',
               (student_name, student_grade))
```

In this approach, we used a **prepared statement**. But what does that actually mean? A prepared statement precompiles the SQL code and safely inserts the values (in this case, the name and grade) into the statement. This is in contrast to using **string interpolation**, where the entire SQL statement, including user input, is constructed and then compiled.

The key advantage of prepared statements is that they separate the SQL logic from the data, ensuring that any input provided by the user is treated strictly as data and not as part of the executable SQL code. This method effectively prevents the user from injecting malicious code into the SQL statement, making it a robust defence against SQL injection attacks.

By using prepared statements, you safeguard your application against the risks associated with allowing user input to directly influence SQL code execution. This is a critical practice in developing secure, resilient systems.

Practical task 1

Follow these steps:

1. Open up your task folder for this task. You will see many different files. However, the only files that you will need to modify are **input1.txt** and **input2.txt**.
2. **input1.txt** and **input2.txt** are inputs to **input1.py** and **input2.py**, respectively. These files, **input1.py** and **input2.py**, create a SQL table called `Student` and populate it with entries. Then, they perform their respective tasks.
    - **input1.txt:** This file searches the database for a student with a particular `first_name`.
    - **input2.txt:** This file logs a user onto the system. The two fields required for login are the email address and student number.
3. However, these Python files have been set up for failure: You can use the input files to inject your own SQL code! Modify the input files in the following manner:
    - **input1.txt:** Inject SQL code to delete all entries from the database.
    - **input2.txt:** Inject SQL code to log yourself into the account using the email address “wimbledon@strange.com”. Do this without specifying the student ID.
4. Complete your inputs, then save and submit your work.

Practical task 2

1. Create two new Python scripts: **safe_input1.py** and **safe_input2.py**. Copy the code from **input1.py** and **input2.py**, and place each copied program into its own respective new file.
2. Now modify the Python code to use **prepared statements** and ensure that your SQL injections do not work.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

SecureFlag. (n.d.). *SQL injection*. **https://knowledge-base.secureflag.com/vulnerabilities/sql_injection/sql_injection_vulnerability.html**