SQL Injection Attacks
SQL injection attacks

- Affect applications that use untrusted input as part of an SQL query to a back-end database
- Specific case of a more general problem: using untrusted input in commands

SQL injection: example

- Consider a browser form, e.g.:

  ![Browser Form](image)

- When the user enters a number and clicks the button, this generates an http request like
  
  https://www.pizza.com/show_orders?month=10
Example continued…

- Upon receiving the request, a script might generate an SQL query as follows:

  ```sql
  sql_query = "SELECT pizza, quantity, order_day "
  + "FROM orders "
  + "WHERE userid=" + session.getCurrentUserId()
  + " AND order_month= "
  + request.getParameter("month");
  ```

- A normal query would look like:

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month=10
  ```

Example continued…

- What if the user makes a modified http request:
  ```
  https://www.pizza.com/show_orders?month=0%20OR%201=1
  ```

  (Parameters transferred in URL-encoded form, where meta-characters are encoded in ASCII)

- This has the effect of setting
  ```
  request.getParameter("month")
  ```
  equal to the string
  ```
  0 OR 1=1
  ```
Example continued

- So the script generates the following SQL query:

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE (userid=4123
  AND order_month=0) OR 1=1
  ```

  - Since AND takes precedence over OR, the above always evaluates to TRUE
  - The attacker gets every entry in the table!

Even worse...

- Craft an http request that generates an SQL query like the following:

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month=0 OR 1=0
  UNION SELECT cardholder, number, exp_date
  FROM creditcards
  ```

  - Attacker gets the entire credit-card table instead!
More damage...

- SQL queries can encode multiple commands, separated by `;`

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0;
DROP TABLE creditcards
```

- Credit-card table deleted!

More damage...

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0;
INSERT INTO admin VALUES ('hacker', ...)
```

- User (with chosen password) entered as an administrator!
  - Database owned!
May need to be more clever...

- Consider the following script for text queries:

  ```java
  sql_query = "SELECT pizza, quantity, order_day "
  + "FROM orders "
  + "WHERE userid=" + session.getCurrentUserId()
  + " AND topping=‘""
  + request.getParameter("topping") + "’"
  ```

- Previous attacks will not work directly, since the commands will be quoted.

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month='0;
  INSERT INTO admin VALUES ('hacker', ...)
  ```

Example continued...

- Craft an http request where `request.getParameter("topping")` is set to `abc'; DROP TABLE creditcards; --`

  - The effect is to generate the SQL query:

    ```sql
    SELECT pizza, quantity, order_day
    FROM orders
    WHERE userid=4123
    AND toppings='abc';
    DROP TABLE creditcards ; --'
    ```

  - (`--` represents an SQL comment)
Second-order SQL injection

- Use a previously stored value to do SQL injection
- E.g., say stored username contains a single quote, encoded appropriately when first stored, e.g.,

```
INSERT INTO users (uname, passwd)
VALUES ('o''connor','terminator')
```
- Later, can change own password:

```
query2 = "UPDATE users SET passwd=" +
         + new_password + " WHERE uname="
         + uname + "";
```
- What if `uname = admin' --` ?

Solutions?
- Defense-in-depth…
  - Use several solutions, as appropriate

- Blacklisting
- Whitelisting
- Sanitization
- Prepared statements/bind variables
- Mitigate the impact of SQL injections
Blacklisting?

- I.e., searching for/preventing ‘bad’ inputs
- E.g., for previous example:

```java
sql_query = "SELECT pizza, quantity, order_day " + "FROM orders " + "WHERE userid=" + session.getCurrentUserId() + " AND topping= ' " + kill_chars(request.getParameter("topping")) + "'
```

- ...where kill_chars() deletes, e.g., quotes and semicolons

Drawbacks of blacklisting

- How do you know if/when you’ve eliminated all possible ‘bad’ strings?
  - If you miss one, could allow successful attack
- Does not prevent first set of attacks (numeric values)
  - Although similar approach could be used, starts to get complex!
- May conflict with functionality of the database
  - E.g., user with name O’Brien
Whitelisting

- Check that user-provided input is in some set of values known to be safe
  - E.g., check that month is an integer in the right range

- If invalid input detected, better to reject it than to try to fix it
  - Fixes may introduce vulnerabilities
  - *Principle of fail-safe defaults*

Prepared statements/bind variables

- **Bind variables**: placeholders guaranteed to be data (not control), in correct format

- **Prepared statements**: allow creation of queries with bind variables
  - Parameters not involved in query parsing
Example (Java)

```java
PreparedStatement ps =
    db.prepareStatement(
        "SELECT pizza, quantity, order_day 
        + "FROM orders WHERE userid=? 
        AND order_month=?"");

ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, 
    Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```

- Query parsed w/o parameters
- Bind variables are typed

Mitigating the impact

- Limit privileges
  - I.e., allow SELECT queries on the orders database, but no queries on creditcards database
  - Can limit commands, or tables to which access is given (or both)
  - Principle of least privilege
  - Not a complete fix, but it helps

- Encrypt sensitive data stored in database
  - E.g., orders in the clear but credit card numbers encrypted