

Prepared by Dr. Nermin Hamza

CYB740

Penetration test and Ethical hacking

Lecture 07



SQL Injection

Lecture 07

CYB740

Penetration test and Ethical hacking

Prepared by
Dr. Nermin Hamza

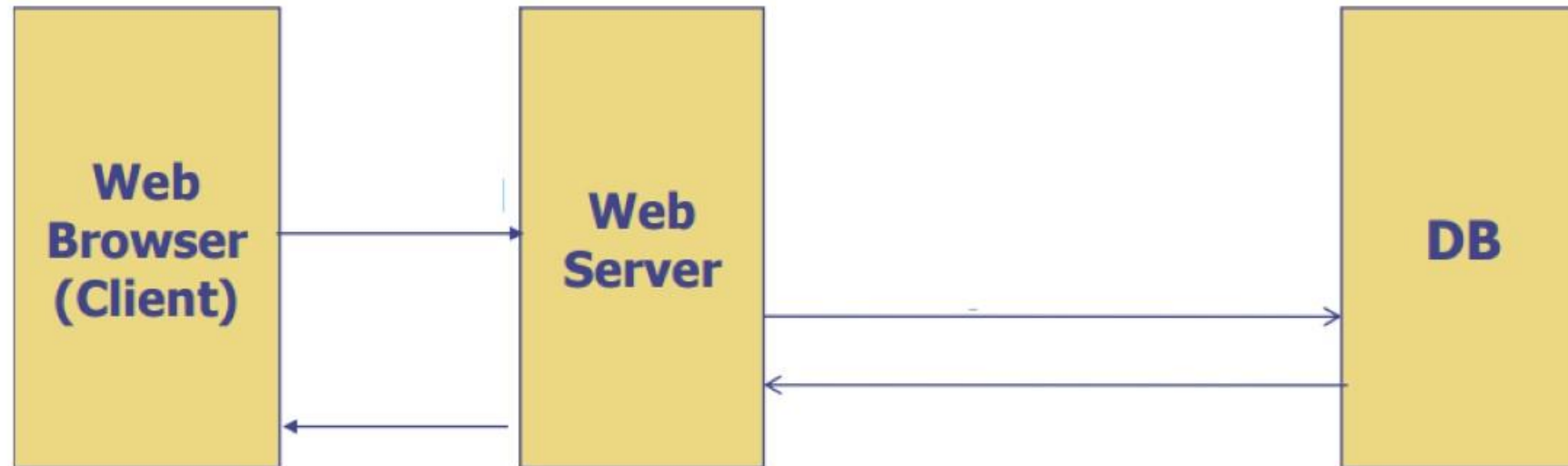
- What is SQL Injection?
- Why : SQL Injection
- Example of SQL Injection
- SQL injection types
- Sources of SQL Injection
- SQL Injection Methodology
- SQL Injection Tool
- SQL Injection Prevention



What is SQL Injection?

- SQL injection is a technique used to take advantage of non-validated input vulnerabilities to pass SQL commands through a web application for execution by a backend database.
- SQL injection is a basic attack used to either gain unauthorized access to a database or to retrieve information directly from the database.
- It is a flaw in web applications and not a database or web server issue.





Normal Query

<?

```
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();
```

```
}
```

```
?>
```

Member Login

Username :

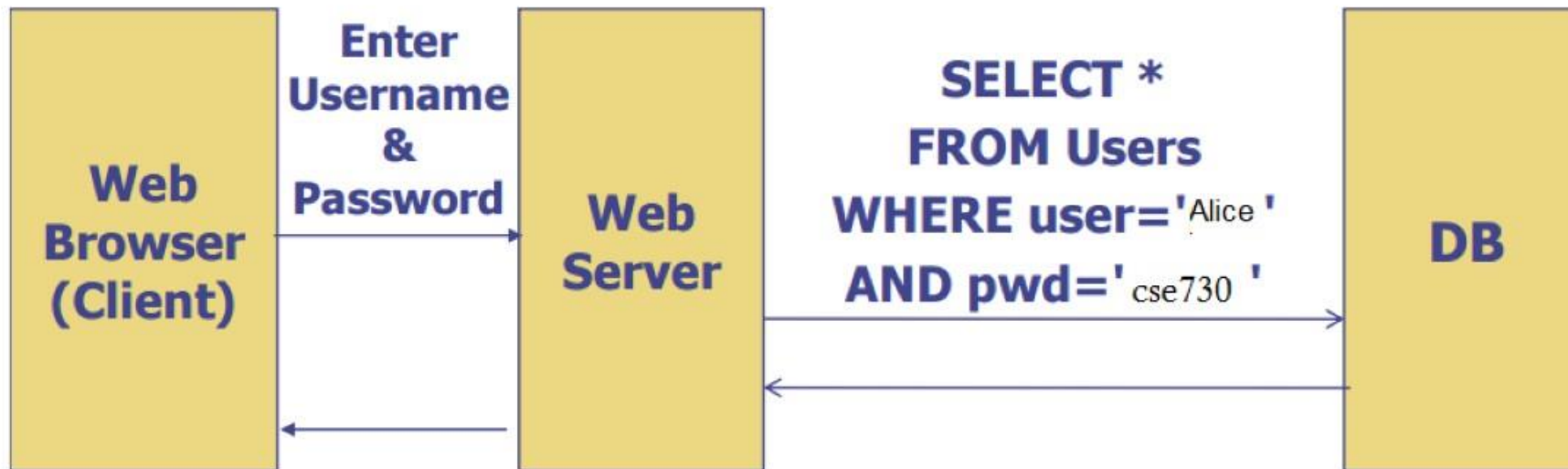
Password :

Member LoginUsername : Password : **user****pass**

Alice

cse730

```
SELECT * FROM `login` WHERE `user` = 'Alice' , and 'pass'  
= ' *****' ;
```



Normal Query

Why Bother about SQL Injection?

On the basis of application used and the way it processes user supplied data, SQL injection can be used to implement the attacks mentioned below:

- **Authentication Bypass:** an attacker logs onto an application without providing valid user name and password and gains administrative privileges.
- **Information Disclosure:** an attacker obtains sensitive information that is stored in the database.

- **Compromised Data Integrity:** An attacker uses this attack to deface a web page, insert malicious content into web pages, or alter the contents of a database.
- **Compromised Availability of Data:** Attackers use this attack to delete the database information, delete log, or audit information that is stored in a database.
- **Remote Code Execution:** It assists an attacker to compromise the host OS

SQL Injection and Server-side Technologies

- **Server-side Technology:** Powerful server-side technologies like ASP.NET and database servers allow developers to create dynamic, data-driven websites with incredible ease.
- **Exploit:** The power of ASP.NET and SQL can easily be exploited by hackers using SQL injection attacks.
- **Susceptible Databases:** All relational databases, SQL Server, Oracle, IBM DB2, and MySQL, are susceptible to SQL-injection attacks.
- **Attack:** SQL injection attacks do not exploit a specific software vulnerability, instead they target websites that do not follow secure coding practices for accessing and manipulating data stored in a relational database.

How Web Applications Work

- A user enters a user name and password that matches a record in the user's table.
- A **dynamically generated SQL query** is used to retrieve the number of matching rows.
- The user is then authenticated and redirected to the requested page.

```
SELECT Count(*) FROM Users WHERE UserName=username AND Password=password
```

- When the attacker enters **blah' or 1=1 --** then the SQL query will look like:

```
SELECT Count(*) FROM Users WHERE UserName='blah' Or 1=1 --'  
AND Password=""
```

- Because a pair of hyphens designate the beginning of a comment in SQL, the query simply becomes:

```
SELECT Count(*) FROM Users WHERE UserName='blah' Or 1=1
```

Hack Me! SQL Injection

' OR 'a'='a

' OR 'a'='a

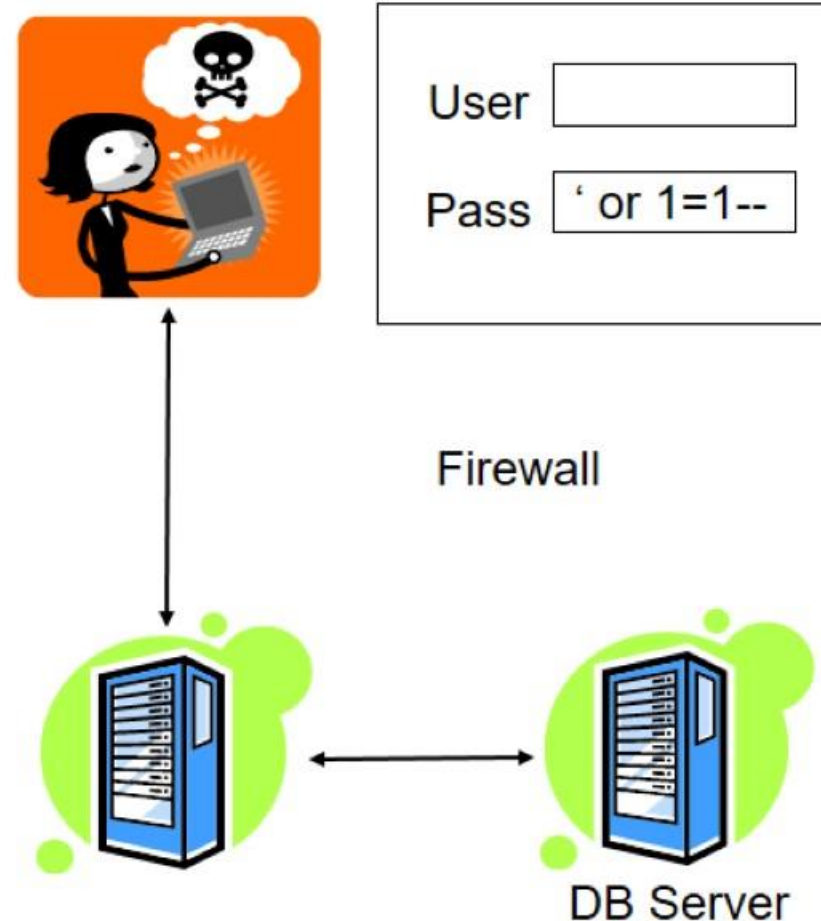
**Member Login**Username : Password :

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

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.



```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

```
DELETE FROM table_name WHERE condition;DELETE  
FROM table_name WHERE condition;
```

```
INSERT INTO table_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);
```

Example of SQL Injection: Updating Table

```
SELECT * FROM `login` WHERE `user`=' user  
inputs ' AND `pass`='user inputs'
```

User inputs: `="; UPDATE `login` SET `pass`='pass123' WHERE `user`='timbo317';--

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

Example of SQL Injection: Adding New Records

User input: `' ; INSERT INTO `login` ('user','pass') VALUES ('haxor','whatever'); --`

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

Example of SQL Injection: Deleting a Table

```
blah'; DROP TABLE Creditcard; --
```



SQL Query Executed

```
SELECT jb-email, jb-passwd, jb-login_id, jb-last_name FROM members  
WHERE jb-email = 'blah'; DROP TABLE Creditcard; --';
```

Hack Me! SQL Injection

```
'; DROP TABLE 'login'; --
```



Member Login

Username : Password :

There are two main types of SQL injection:

- **Error-Based SQL Injection:**
 - Attackers intentionally insert bad input into an application, causing it to throw database errors.
 - The attacker reads the database-level error messages that result in order to find an SQL injection vulnerability in the application.
 - Based on this, the attacker then injects SQL queries that are specifically designed to compromise the data security of the application.
- **Blind SQL Injection:**
 - The attacker has no error messages from the system with which to work.
 - Instead, the attacker simply sends a malicious SQL query to the database.

Error Based SQL Injection (?)

- Error based SQL Injection forces the database to perform some operation in which the **result will be an error**.
- This exploitation may differ from one DBMS to the other.

```
Microsoft OLE DB Provider for  
ODBC Drivers error '80040e14'  
[Microsoft][ODBC SQL Server  
Driver][SQL Server]Unclosed  
quotation mark before the  
character string ''.  
/shopping/buy.aspx, line 52
```

Error Based SQL Injection:

- UNION SQL Injection
- System Stored Procedure
- PiggyBacked Queries:
- Tautology
- Illegal/Logically Incorrect Query

SQL Injection types

Union SQL Injection

- The **UNION** operator is used to combine the result-set of two or more **SELECT** statements.
- Every **SELECT** statement within **UNION** must have the same number of columns
- The columns must also have similar data types
- The columns in every **SELECT** statement must also be in the same order

```
SELECT column_name(s) FROM table1  
UNION  
SELECT column_name(s) FROM table2;
```

Union SQL Injection

- This technique involves joining a forged query to the original query.
- Result of forged query will be joined to the result of the original query thereby allowing to obtain the values of fields of other tables.

- **Example:**

SELECT Name, Phone, Address FROM Users WHERE Id=\$id

- Now set the following Id value:

\$id=1 UNION ALL SELECT creditCardNumber, 1, 1 FROM CreditCardTable

- The final query is as shown below:

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber, 1, 1 FROM CreditCardTable

The above query joins the result of the original query with all the credit card users.

USER INPUT (MYSQL SYSTEM TABLE).

```
1 UNION SELECT 1 FROM information_schema.tables
```

QUERY GENERATED.

```
SELECT name, description, price FROM products WHERE  
category=1 UNION SELECT 1 FROM information_schema.tables
```

ERROR RETURNED.

```
ORA-00942: table or view does not exist.
```

This table name is not valid. Let's try with system tables of another popular DBMS.

USER INPUT (ORACLE SYSTEM TABLE).

```
1 UNION SELECT 1 FROM all_tables
```

QUERY GENERATED.

```
SELECT name, description, price FROM products WHERE  
category=1 UNION SELECT 1 FROM all_tables
```

ERROR RETURNED.

```
ORA-01789: query block has incorrect number of result  
columns.
```

An error is still returned but **it is not related to an invalid table name**. The attacker now knows a valid table and he can solve the problem related to the number of columns.

PiggyBacked Queries:

- An attacker injects additional queries into the original query to extract data, add or modify data, or execute remote commands.
- In this scenario, the attacker does not intend to modify the original intended query but to include new queries that piggy-back on the original query.
- As a result, the DBMS receives multiple SQL queries. The first is the normal query which is executed normally, while the subsequent ones are executed to satisfy the attack.
- Here is a common form using a query delimiter (;) for this attack:
normal SQL statement + ";" + INSERT (or UPDATE, DELETE, DROP) <rest of injected query>

Stored Procedures:

- When a normal SQL statement (i.e., SELECT) is created as a stored procedure, an attacker can inject another stored procedure as a replacement for a normal stored procedure to performing privilege escalation, execute remote commands.
- Here is a common form using a query delimiter (;) and the "SHUTDOWN" store procedure for this attack:

normal SQL statement + "; SHUTDOWN; " <rest of injected query>

Tautologies:

- An attacker injects a query that always evaluates to true for entries in the database to bypass authentication, identify injectable parameters, or extract data.
- For example:

Username known Input (username): `jdope' or '1'='1--`

Sql: `SELECT * FROM students WHERE username = 'jdope' or '1'='1' -- AND password =`

Result: All students are retrieved.

Both username and password not known

Input (username): `' or '' = '`

Input (password): `' or '' = '`

Sql: `select * from students where username = '' or '' = '' and password = '' or '' = ''`

Result: All students are retrieved.

Illegal/Logically Incorrect Queries:

- An attacker gathers important information about the type and structure of the back-end database of a Web application by injecting illegal or logically incorrect SQL syntax which will make the application return default error pages that often reveal vulnerable/injectable parameters to the attacker.
- This attack is considered as a preliminary, **information-gathering** step for other SQL injection attacks.

For example:

Probing column name

Input (username): 'ddd'"

Sql: **SELECT * FROM students WHERE username = 'ddd'" AND password =**

Result: "Incorrect syntax near 'ddd'. Unclosed quotation mark after the character string " AND Password='aaa'"."

- **Blind SQL Injection** is used when a web application is vulnerable to an SQL injection, but **the results of the injection are not visible to the attacker.**
- The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page.



Oops!

We are unable to process your request. Please try back later.

Blind SQL Injection

- **Time-intensive:** This type of attack can become **time-intensive because a new statement** must be crafted for each bit recovered.
- **Note:** An attacker can still steal data by asking a series of True and False questions through SQL statements.
- `SELECT * FROM products WHERE id = 1' AND sleep(10);- -`
- If the database server is vulnerable to time-based blind SQL injections, the attacker will see a 10-second delay in the application.

- Injection through user input
 - Malicious strings in web forms.
- Injection through cookies
 - Modified cookie fields contain attack strings.
- Injection through server variables
 - Headers are manipulated to contain attack strings.
- Second-order injection
 - Trojan horse input seems fine until used in a certain situation

```
SELECT * from users where id = 123
```

```
$user = base64_decode($user);
$user = explode(":", $user);
$uid = "$user[0]";
$pwd = "$user[2]";
} else {
$uid = "$user[0]";
$pwd = "$user[2]";
}
if ($uid != "" AND $pwd != "") {
$ssql = "SELECT user_password FROM ".$user_prefix."_users WHERE user_id='$uid'";
```

SQL Injection Methodology

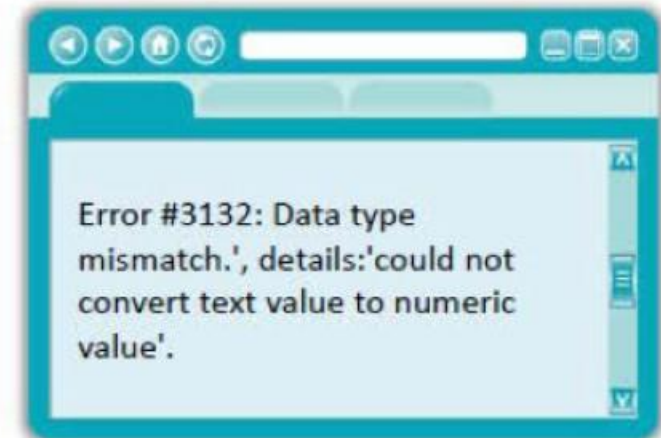
- Information gathering SQL injection and vulnerability detection
- Launch injection attack

Information gathering SQL injection and vulnerability detection

- Inject code into the input fields to generate error messages
- ***Determine database engine*** : most error message will show DB engine

- **Determine the column** : try to insert string into numeric fields the error message will show the data that could not get converted

```
' union select 1,1,'text',1,1,1 --  
' union select 1,1, bigint,1,1,1 --
```



- **Use time delay or Error signature to determine extract information**

```
'; if condition  waitfor delay '0:0:5' --  
'; union select if( condition , benchmark (100000, sha1('test')), 'false' ),1,1,1,1;
```

Oracle

- SYS.USER_OBJECTS
- SYS.TAB, SYS.USER_TABLES
- SYS.USER_VIEWS
- SYS.ALL_TABLES
- SYS.USER_TAB_COLUMNS
- SYS.USER_CATALOG

**MS
Access**

- MsysACEs
- MsysObjects
- MsysQueries
- MsysRelationships

**MySQL**

- mysql.user
- mysql.host
- mysql.db

**MS SQL
Server**

- sysobjects
- syscolumns
- systypes
- sysdatabases



Database features



	MySQL	MSSQL	MS Access	Oracle	DB2	PostgreSQL
String Concatenation	concat(,) concat_ws(delim,)	''+'''	""&""	'' '	"concat " ""+" " '' ''	'' ''
Comments	-- and /**/ and #	-- and /*	No	-- and /*	--	-- and /*
Request Union	union	union and ;	union	union	union	union and ;
Sub-requests	v.4.1 >=	Yes	No	Yes	Yes	Yes
Stored Procedures	No	Yes	No	Yes	No	Yes
Availability of information schema or its Analogs	v.5.0 >=	Yes	Yes	Yes	Yes	Yes

- BSQLHacker
 - Blind SQL Hacker is automated SQL injection tool designed to exploit SQL injection vulnerabilities virtually in any database
- MarathonTool
- SQL Power Injector

SQL Injection Prevention

- Prepared Statements (with Parameterized Queries)
- Character Escaping
- Input Validation
- Password hashing

Prepared Statements (with Parameterized Queries)

- Prepared statements are easy to learn and use, and eliminate the problem of SQL injection. They force you to define SQL code, and pass each parameter to the query later, making a strong distinction between code and data.
- If an attacker supplies a malicious string : for example providing John' or 1=1 for a username, the prepared statement will evaluate this as a literal string. It will look for a user named John' or 1=1 (and fail, because no such user exists) instead of evaluating this statement as code.

```
public boolean login(String username, String password) {  
    PreparedStatement stmt =  
        connection.prepareStatement("SELECT * FROM users WHERE username = ? AND password = ?")  
    stmt.setString(1, username);  
    stmt.setString(2, password);  
    ResultSet rs = stmt.executeQuery();  
  
    if(result.next()) {  
        /* Login Success */  
        return true;  
    }  
    else. {  
        /* Login Failed */  
        return false;  
    }  
}
```

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

Character Escaping

- Character escaping is an effective way of preventing SQL injection.
- Special characters like “/ — ;” are interpreted by the SQL server as a syntax and can be treated as an SQL injection attack when added as part of the input.
- Pre-cleansing input data by “escaping” characters tells SQL values that they should be treated as a string, rather than as a command.
- Escaping special characters doesn't guarantee SQL injection prevention, but it reduces the risk significantly.

```
select * from users where username = 'johndoe\' OR 1==1 --' and password = 'easypassword';
```

ACME Corporation

Username:

Password:

OK

Signup

Input Validation

- Validating input plays a significant role in preventing SQL injection. Suspicious inputs are filtered prior to submission or processing by the server when validated. An example of input validation is an email validator.

Javascript Code to Validate an Email ID

```
function ValidateEmail(mail) { if (/^\w+([\.-]?\w+)*@\w+([\.-]  
]? \w+)*(\.\w{2,3})+$/ .test(myForm.emailAddr.value))  
{ return (true) }
```

```
alert("You have entered an invalid email address!")  
return (false) }
```

Password hashing

- Hashing the user password before entering it into the query is a coincidental security feature to prevent SQL injection, but you can't necessarily do that with all user input. If I'm looking up a Customer by their Name and I have a query like
- `Select * from Customer Where Name like '%userInput%'`

<https://www.sqlinjection.net/union/>

Thank You