Java Script Vulnerabilities

* Exploiting Javascript vulnerabilities can manipulate Data, redirect sessions, modify and steal data, and much more.

Although Javascript is typically thought of as a client-side application,
Javascript security issues can create problems on server-side environments as well

* What are common Java Script vulnerabilities?

* most common Javascript attack vectors include
→ executing malicious scripts,
→ stealing a user's established session data (or) data from browser's Local Storage,
→ tricking users into performing unintended actions and exploiting vulnerabilities in source code of web applications.

* Common Javascript Security Vulnerabilities

Cross-site Scripting (XSS)
Cross-site request forgery (CSRF)
Third-party security vulnerabilities
SQL Injection
Sensitive cookie exposure

Cross-site Scripting (XSS):

* is one of the most widespread security issues in web applications.
* It occurs when an attacker injects malicious code into the client-side of the application.
* This normally happens when an application accepts untrusted (or user-supplied) data on a webpage without escaping or validating it properly.

→ If an attacker exploits XSS vulnerabilities, they could perform malicious actions like tampering, data theft, remote control, or even malware distribution.

To prevent XSS attacks, developers should separate untrusted data or user input from active browser content.

In JavaScript you can achieve this by

- Validating and sanitizing input from users to ensure it only contains acceptable characters that can not be used to launch XSS attacks.
- Using safe methods such as innerText for manipulating DOM.
  Unlike innerHTML, this method escapes dangerous content, thereby preventing DOM-based XSS.
- Using frameworks that automatically escape XSS vulnerabilities by design.

In the example above, the application gets user IDs from URLs and returns the corresponding email address by querying the database.

Two things are wrong in the code snippet

* First thing the db query is built using a String concatenation.

The second issue is that the user input is connected to the query instead of being handled as untrusted data.

* To mitigate SQL injections, developers should always perform proper input validation.

→ when input from the user fails the validation checks, the SQL query is not executed.

→ Another way of preventing SQL injection is using parameterized queries (or) prepared statements instead of concatenations

Example:

```
// Wrong implementation
// unvalidated "clientName" parameter appended
to the query allows an attacker to inject any SQL
Code.

String query = "SELECT account_balance from
User_Data WHERE user_name = "
        + request.getParameter("clientNa
```

how to use xss-filters

Example:

```
Const express = require('express');
Const xssFilters = require('xss-filters');
Const Util = require('util');
Const app = express();
app.get('/', (req, res) => { Const unsafeFirstName =
  req.query.firstName;
Const SafeFirstName =
  xssFilters.inHTMLData(unsafeFirstName);
  res.Send(Util.format('<h1>Tom's <h1>',
                       SafeFirstName)); });
```

SQL injection:

SQL database are vulnerable to injection attacks where query parameters are exploited to execute arbitrary instructions.

```
Const express = required('express');
Const db = require('./db');

Const router = Express.Router();
router.get('/email', (req, res) => {
  db.query('SELECT email from users where
  id=' + req.query.id);
      .then((record) => {
        //logical flow
        res.send(record[0]);

  });
});
```

In the example above, the application gets UserID's from URL's and ~~returns~~ retrieves the corresponding email address by querying the database.

Two things are wrong in the code snippet

* First thing the db query is built using a String concatenation.

The second issue is that the user input is connected to the query instead of being handled as untrusted data.

* To mitigate SQL injections, developers should always perform proper input validation.

-> when input from the user fails the validation checks, the SQL query is not executed.

-> Another way of preventing SQL injection is using parameterized queries (or) prepared statements instead of concatenations

Example:

```
// Wrong implementation
// unvalidated "clientName" parameter appended
to the query allows an attacker to inject any SQL
code.

String query = "SELECT account_balance from
User_Data WHERE user_name = "
        + request.getParameter("clientNa
```

try {
    Statement statement = Connection.CreateStatement(
    ...);
    ResultSet results = Statement.executeQuery(query);
}

//Correct implementation.

String clientName =
        request.getParameter("clientName");

// validate input to detect attacks.

String query = "SELECT account_balance FROM
        user_data WHERE user_name = ?";

PreparedStatement pstmt = Connection.
        preparedStatement(query);

pstmt.setString(1, clientname);
ResultSet results = pstmt.ExecuteQuery();

Sensitive Cookie Exposure :

* The client side script on every browser can access all the content returned by application to the server
    → This includes cookies that often contain sensitive data such as session IDs
    → Exposing session identifiers, whether in URL, error messages, or logs is a bad practice that opens up an application to security issues like

Cross-site request forgery (CSRF), Session hijacking and Session Fixation

→ To prevent this, developers must Consistently use Https's and implement Http-only cookies.
→ The Http-only attribute in cookies tells the browser to prevent cookie access through the DOM.
→ By doing this client-side script attacks are prevented from accessing sensitive data stored in Cookies

→ Another way of saving user sessions is opting by per-requests as opposed to using per-session Identifiers.

Putting all together:

Adopting good coding practise can secure applications against common Javascript Vulnerabilities on both the client-side and server-side.

When using Java script follow the key guidelines

* Never trust user input
* Use proper encoding/escaping
* Sanitize user input
* Define a Content Security Policy
* Set Secure Cookies
* Secure API keys on client-side
* Encrypt data transmitted between the client and server
* Use Secure Components and APIs