# Owasp top10 secure coding and How to do Source code review NOtes...

# Broken Access Control in Express.js

Bug Bounty Hunter

March 29, 2025

## 1 Introduction

Alright, bhai! Let's dive into this topic step-by-step. Since you're a source code reviewer, bug bounty hunter, aur certified ethical hacker, we'll make sure you get solid examples of **vulnerable code** aur **secure code** in Express.js, specifically for **Broken Access Control** from the OWASP Top 10. I'll explain everything in Hinglish so it's easy to samajh and you can use it in your job to find flaws aur secure websites like a pro.

## 2 Broken Access Control Kya Hai?

Broken Access Control hota hai jab koi user ya attacker aisa data ya functionality access kar leta hai jiska usko permission nahi hona chahiye. For example, ek normal user admin ka dashboard dekh le ya kisi aur user ka private data edit kar de. Yeh problem tab aati hai jab code mein proper checks nahi hote.

**Chhota Example:**

- Ek website pe `/user/profile` endpoint hai jo sirf logged-in user ka data dikhana chahiye.
- Lekin agar koi attacker `/user/profile?userId=123` mein `userId` change karke dusre user ka data dekh le, toh yeh Broken Access Control hai.

Ab main tumhe **vulnerable code** aur **secure code** side-by-side dikhaunga Express.js mein, with explanations.

## 3 Example 1: User Profile Access

### 3.1 Vulnerable Code:

Listing 1: Vulnerable User Profile Endpoint

```
const express = require('express');
const app = express();

app.get('/user/profile', (req, res) => {
  const userId = req.query.userId; // Attacker yahan userId change kar sakta hai
  // Database se data fetch kar rahe hain bina check ke
  const userData = { id: userId, name: "Test User", email: "test@example.com" };
  res.json(userData);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 3.2 Kyun Vulnerable Hai?

- Yeh code `req.query.userId` direct le raha hai aur koi check nahi kar raha ki current logged-in user ka ID match karta hai ya nahi.
- **Attacker** bas URL mein `?userId=999` daal kar kisi bhi user ka data dekh sakta hai.
- No authentication ya authorization check hai.

## 3.3 Secure Code:

Listing 2: Secure User Profile Endpoint

```
const express = require('express');
const app = express();

// Dummy function for logged-in user
const getCurrentUser = (req) => {
  return { id: 1, name: "Logged User" }; // Yeh session ya token se aayega real mein
};

app.get('/user/profile', (req, res) => {
  const currentUser = getCurrentUser(req); // Current logged-in user ka data
  const requestedUserId = parseInt(req.query.userId);

  // Check karo ki requested userId current user se match karta hai ya nahi
  if (currentUser.id !== requestedUserId) {
    return res.status(403).json({ error: 'Access denied, bhai!' });
  }

  const userData = { id: requestedUserId, name: "Test User", email: "test@example.com" };
  res.json(userData);
```

```
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 3.4 Kyun Secure Hai?

- Ab code check karta hai ki jo `userId` request mein aaya hai, woh current logged-in user ka ID hai ya nahi.
- Agar match nahi karta, toh **403 (Forbidden)** response bhejta hai.
- Isse attacker dusre user ka data nahi dekh sakta.

## 4 Example 2: Admin Dashboard Access

### 4.1 Vulnerable Code:

Listing 3: Vulnerable Admin Dashboard

```
const express = require('express');
const app = express();

app.get('/admin/dashboard', (req, res) => {
  // Koi role check nahi hai, bas endpoint hit karne se admin dashboard
      dikh raha hai
  res.send('Welcome to Admin Dashboard, sab kuch dikhta hai!');
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 4.2 Kyun Vulnerable Hai?

- Yeh code kisi bhi user ko `/admin/dashboard` pe jaane deta hai bina yeh check kiye ki woh admin hai ya nahi.
- **Attacker** bas URL hit karke admin panel access kar sakta hai.

### 4.3 Secure Code:

Listing 4: Secure Admin Dashboard with Middleware

```
const express = require('express');
const app = express();

// Dummy function for logged-in user
```

```
const getCurrentUser = (req) => {
  return { id: 1, role: "user" }; // Real mein yeh session ya JWT se
      aayega
};

const isAdmin = (req, res, next) => {
  const currentUser = getCurrentUser(req);
  if (currentUser.role !== 'admin') {
    return res.status(403).json({ error: 'Tu admin nahi hai, bhai!' });
  }
  next(); // Agar admin hai toh aage badho
};

app.get('/admin/dashboard', isAdmin, (req, res) => {
  res.send('Welcome to Admin Dashboard, sirf admin ke liye!');
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 4.4 Kyun Secure Hai?

- `isAdmin` middleware check karta hai ki user ka role "admin" hai ya nahi.
- Agar nahi hai, toh **403 error** bhejta hai.
- Sirf admin role wale users hi dashboard dekh sakte hain.

## 5 Example 3: Rate Limiting (Broken Access Control Prevention)

### 5.1 Vulnerable Code:

Listing 5: Vulnerable Login Endpoint

```
const express = require('express');
const app = express();

app.post('/login', (req, res) => {
  const { username, password } = req.body;
  // Koi rate limit nahi, attacker infinite requests bhej sakta hai
  if (username === 'admin' && password === 'pass123') {
    res.json({ message: 'Login successful' });
  } else {
    res.status(401).json({ error: 'Galat credentials' });
  }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 5.2 Kyun Vulnerable Hai?

- Isme koi rate limiting nahi hai, toh attacker **brute force attack** kar sakta hai aur infinite login attempts bhej sakta hai.
- Yeh Broken Access Control ko allow karta hai kyunki koi protection nahi hai against excessive requests.

## 5.3 Secure Code:

Listing 6: Secure Login with Rate Limiting

```
const express = require('express');
const rateLimit = require('express-rate-limit'); // Rate limiting ke
    liye package
const app = express();

// Rate limit setup: 5 requests per IP in 15 minutes
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // 5 requests allowed
  message: 'Bhai, bahut zyada requests bhej raha hai, thodi der ruk!'
});

app.post('/login', loginLimiter, (req, res) => {
  const { username, password } = req.body;
  if (username === 'admin' && password === 'pass123') {
    res.json({ message: 'Login successful' });
  } else {
    res.status(401).json({ error: 'Galat credentials' });
  }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 5.4 Kyun Secure Hai?

- `express-rate-limit` package use kiya gaya hai jo har IP ke liye 15 minute mein sirf 5 requests allow karta hai.
- Agar attacker zyada requests bhejta hai, toh woh block ho jayega aur **"Too Many Requests"** message milega.
- Yeh brute force attacks ko rokta hai aur access control ko strong banata hai.

# 6 Broken Access Control Ko Rokne Ke Methods

- **Authentication & Authorization:**

  - Har sensitive endpoint pe check karo ki user authenticated hai aur uska role ya permission sahi hai.
  - Example: `isAdmin` middleware jaise upar use kiya.

- **Rate Limiting:**
  - `express-rate-limit` jaise tools se limit lagao taaki attacker excessive requests na bhej sake.

- **CAPTCHA:**
  - Login ya sensitive actions pe CAPTCHA add karo taaki bots automate na kar sake.
  - Example: Google reCAPTCHA integrate kar sakte ho.

- **IP-Based Rate Limiting:**
  - Har IP ke liye request count track karo aur limit exceed hone pe block kar do (upar wala code dekho).

- **Session Management:**
  - Valid session ya JWT token check karo har request pe.

# 7 Rate Limit Attack Ko Express.js Mein Kaise Bachayein?

- **Step 1:** `express-rate-limit` install karo:

```
npm install express-rate-limit
```

- **Step 2:** Upar wale secure code jaise configure karo:
  - `windowMs`: Time window define karo (e.g., 15 minutes).
  - `max`: Max requests allowed in that window.
  - Custom message ya status code set karo.

- **Step 3:** Redis ya database ke saath integrate karo agar distributed system hai, taaki memory-based limit crash na kare.

**Example Output:**

- Agar koi 5 se zyada requests bhejta hai 15 minute mein, toh response aayega:

```
{ "message": "Bhai, bahut zyada requests bhej raha hai, thodi der ruk!" }
```

# Cryptographic Failures in Express.js

## 8 Introduction

Alright, bhai! Let's dive into **Cryptographic Failures** from the OWASP Top 10. Main tujhe is topic ko Hinglish mein samjhaunga, with vulnerable aur secure code examples in Express.js, aur har cheez ko break down karke bataunga taaki tu apne job mein source code review aur ethical hacking mein pro ban jaye. Topic hai **Cryptographic Failures**, aur isme weak algorithms, improper key management, aur inadequate encryption practices cover karenge. Chalo shuru karte hain!

## 9 Cryptographic Failures Kya Hai?

Cryptographic Failures tab hote hain jab aapka code sensitive data ko protect karne mein fail ho jata hai kyunki aap weak ya outdated encryption use kar rahe ho, keys ko galat tarike se manage kar rahe ho, ya data ko properly encrypt nahi kar rahe ho. Isse attacker data ko crack kar sakta hai, eavesdrop kar sakta hai, ya unauthorized access le sakta hai.

**Examples:**

- Agar tum MD5 ya SHA-1 jaise purane algorithms use karte ho, toh attacker inhe tod sakta hai.
- Agar keys code mein hard-coded hain ya insecure jagah pe store hain, toh koi bhi unhe chura sakta hai.
- Agar data encryption weak hai ya transmission ke waqt encrypt nahi hota, toh data leak ho sakta hai.

Ab main tujhe **vulnerable code** aur **secure code** dikhaunga, with reasons.

## 10 Example 1: Weak Cipher Usage

### 10.1 Vulnerable Code:

Listing 7: Vulnerable Encryption Using DES

```
const express = require('express');
const crypto = require('crypto');
const app = express();

app.post('/encrypt', (req, res) => {
  const data = req.body.secret;
```

```
  const key = 'mysecretkey12345'; // Weak key
  const cipher = crypto.createCipher('des', key); // DES is outdated aur weak
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  res.json({ encrypted });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 10.2 Kyun Vulnerable Hai?

- Yeh code **DES** (Data Encryption Standard) use kar raha hai, jo ab outdated aur weak mana jata hai kyunki iski key size chhoti hai (56-bit) aur modern hardware se easily crack ho sakta hai.
- Key bhi weak hai (`mysecretkey12345`) aur predictable hai.
- **Attacker** is encrypted data ko brute force ya cryptographic attacks se decrypt kar sakta hai.

## 10.3 Secure Code:

Listing 8: Secure Encryption Using AES-256

```
const express = require('express');
const crypto = require('crypto');
const app = express();

app.post('/encrypt', (req, res) => {
  const data = req.body.secret;
  const key = crypto.randomBytes(32); // 256-bit strong key
  const iv = crypto.randomBytes(16); // Initialization vector for randomness
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv); // AES-256 strong cipher
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  res.json({ encrypted, iv: iv.toString('hex'), key: key.toString('hex') }); // IV aur key bhej rahe hain (real mein secure storage mein rakhna)
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 10.4 Kyun Secure Hai?

- **AES-256-CBC** use kiya jo ek modern aur strong encryption algorithm hai.

- Key 256-bit ka hai aur `crypto.randomBytes` se randomly generate kiya gaya hai, toh predictable nahi hai.
- **IV (Initialization Vector)** add kiya gaya taaki har encryption unique ho, even agar same data ho.
- Yeh brute force aur cryptographic attacks ke against resistant hai.

# 11 Example 2: Hardcoded Keys

## 11.1 Vulnerable Code:

Listing 9: Vulnerable Hardcoded Key

```
const express = require('express');
const crypto = require('crypto');
const app = express();

const SECRET_KEY = 'hardcoded123'; // Hard-coded key, bhai yeh galat hai!

app.post('/encrypt', (req, res) => {
  const data = req.body.secret;
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(SECRET_KEY), iv);
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  res.json({ encrypted });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 11.2 Kyun Vulnerable Hai?

- **Hard-coded key** (`SECRET_KEY`) code mein likha hai, jo kisi bhi developer ya attacker ko source code dekhne pe mil jayega.
- Key chhota aur weak hai (`hardcoded123`), toh guess karna ya brute force karna aasan hai.
- Agar key leak ho gaya, toh saara encrypted data compromise ho jayega.

## 11.3 Secure Code:

Listing 10: Secure Key Management Using .env

```
const express = require('express');
const crypto = require('crypto');
require('dotenv').config(); // .env file se key load karne ke liye
const app = express();

app.post('/encrypt', (req, res) => {
  const data = req.body.secret;
  const key = Buffer.from(process.env.SECRET_KEY, 'hex'); // .env se
     strong key load karo
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  res.json({ encrypted, iv: iv.toString('hex') });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 11.4 Setup for .env:

- `npm install dotenv` karo.
- `.env` file banao aur usme likho:

```
SECRET_KEY=your_random_32_byte_key_in_hex_here
```

(Key generate karne ke liye: `crypto.randomBytes(32).toString('hex')`)

### 11.5 Kyun Secure Hai?

- Key ab hard-coded nahi hai, balki **.env** file se load ho raha hai, jo source code mein nahi dikhta.
- Key strong hai (32 bytes = 256-bit) aur randomly generated hai.
- `.env` file ko gitignore karo taaki repo mein upload na ho, aur production mein secure vault (jaise AWS Secrets Manager) use karo.

## 12 Example 3: Securing Hashing Algorithm

### 12.1 Vulnerable Code:

Listing 11: Vulnerable MD5 Hashing

```
const express = require('express');
const crypto = require('crypto');
```

```
const app = express();

app.post('/register', (req, res) => {
  const { password } = req.body;
  const hash = crypto.createHash('md5').update(password).digest('hex');
      // MD5 weak hai
  res.json({ message: 'User registered', hash });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 12.2 Kyun Vulnerable Hai?

- **MD5** use kiya gaya jo ek outdated aur insecure hashing algorithm hai.
- MD5 mein collision attacks possible hain (2 alag inputs se same hash ban sakta hai).
- Yeh fast hai, toh brute force karna aasan hai, aur passwords ko crack karna simple ho jata hai.

## 12.3 Secure Code:

Listing 12: Secure Password Hashing with Bcrypt

```
const express = require('express');
const bcrypt = require('bcrypt');
const app = express();

app.post('/register', async (req, res) => {
  const { password } = req.body;
  const saltRounds = 10;
  const hash = await bcrypt.hash(password, saltRounds); // Bcrypt
      strong hai
  res.json({ message: 'User registered', hash });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 12.4 Setup:

- `npm install bcrypt` karo.

## 12.5 Kyun Secure Hai?

- **Bcrypt** use kiya jo password hashing ke liye specially designed hai.

- Yeh salt automatically add karta hai, toh har baar same password ka alag hash banta hai.
- Computationally intensive hai, toh brute force attacks mushkil ho jate hain.

# 13 Cryptographic Failures Prevention Guide

- **Strong Encryption:**
  - AES-256 jaise modern algorithms use karo, purane jaise DES ya MD5 avoid karo.
  - IV (Initialization Vector) use karo taaki encryption unique rahe.
- **Secure Key Management:**
  - Keys ko hard-code mat karo, `.env` ya secure vaults (AWS Secrets Manager, HashiCorp Vault) mein store karo.
  - Keys ko regularly rotate karo (har 6 mahine ya saal mein).
- **Hashing Ke Liye Bcrypt:**
  - Passwords ke liye **bcrypt** use karo kyunki yeh salt add karta hai aur slow hai, jo security ke liye acha hai.
- **Random Values:**
  - Secure random values ke liye `crypto.randomBytes()` ya `secrets.token_hex()` use karo, `random.randint()` avoid karo kyunki woh predictable hai.

# 14 Algorithms Explained in Hinglish

- **hashlib.sha256(), hashlib.sha3_256(), hashlib.sha3:**
  - Yeh cryptographic hash functions hain jo data integrity check ke liye theek hain, lekin passwords ke liye nahi kyunki inme salt nahi hota aur fast hote hain.
  - SHA-256 aur SHA-3 abhi secure hain, lekin bcrypt se kam strong password hashing ke liye.
- **bcrypt:**
  - Password hashing ke liye best hai kyunki yeh salt add karta hai aur slow hai, toh brute force mushkil ho jata hai.
  - Har baar same password ka alag hash banata hai.
- **secrets.token_hex(x):**

  - Yeh Node.js ke `crypto` module ka part hai, secure random tokens banata hai (e.g., `crypto.randomBytes(16).toString('hex')`).
  - Yeh keys ya tokens ke liye acha hai.

- **Insecure Methods:**
  - `hashlib.md5()`: Yeh purana hai aur collision attacks ka shikaar ho sakta hai (2 alag inputs se same hash). Passwords ya signatures ke liye bilkul mat use karo.
  - `hashlib.sha1()`: Yeh bhi outdated hai, collisions possible hain, aur ab secure protocols mein deprecated hai.
  - `random.randint()`: Yeh cryptographic random nahi hai, predictable values deta hai, toh salt ya keys ke liye bekaar hai.

# Injections

## 15 Injections Kya Hai?

Injections tab hote hain jab attacker user input ke through malicious code daal deta hai jo backend pe execute ho jata hai. Yeh sensitive data chura sakta hai, system ko hack kar sakta hai, ya pura database delete kar sakta hai. Common injections hain:

- **SQL Injection**: Database queries mein malicious SQL daal dena.
- **Command Injection**: System commands execute karwana.
- **XML/LDAP Injection**: XML ya LDAP queries ko manipulate karna.

**Example**: Agar ek login form mein user input `username` ya `password` ke jagah `' OR 1=1 --` daal de aur yeh query mein chale jaye, toh bina password ke login ho sakta hai.

## 16 Example 1: SQL Injection

### 16.1 Vulnerable Code:

Listing 13: Vulnerable SQL Query

```
1 const express = require('express');
2 const mysql = require('mysql');
3 const app = express();
4
```

```
const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mydb'
});

app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const query = `SELECT * FROM users WHERE username = '${username}' AND
      password = '${password}'`; // Direct string concatenation
  db.query(query, (err, results) => {
    if (err) throw err;
    if (results.length > 0) {
      res.json({ message: 'Login successful' });
    } else {
      res.status(401).json({ error: 'Galat credentials' });
    }
  });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 16.2 Kyun Vulnerable Hai?

- Yeh code user input (`username` aur `password`) ko direct query mein concatenate kar raha hai.
- Agar attacker `username` mein `' OR 1=1 --` daal de, toh query banegi:

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

- Yeh hamesha true hoga kyunki `1=1` hamesha true hota hai, aur `--` baaki query ko comment kar deta hai. Attacker bina password ke login kar lega.

## 16.3 Secure Code:

Listing 14: Secure Parameterized Query

```
const express = require('express');
const mysql = require('mysql');
const app = express();

const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mydb'
});

app.post('/login', (req, res) => {
```

```
  const { username, password } = req.body;
  const query = 'SELECT * FROM users WHERE username = ? AND password =
     ?'; // Parameterized query
  db.query(query, [username, password], (err, results) => {
    if (err) throw err;
    if (results.length > 0) {
      res.json({ message: 'Login successful' });
    } else {
      res.status(401).json({ error: 'Galat credentials' });
    }
  });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 16.4 Kyun Secure Hai?

- **Parameterized query** use ki gayi hai (`?` placeholders ke saath), jo user input ko query string mein mix hone se rokti hai.
- Input ab SQL code ke roop mein execute nahi hota, balki sirf data ke roop mein treat hota hai.
- `' OR 1=1 --` daalne se bhi query manipulate nahi hogi.

## 17 Example 2: Command Injection

### 17.1 Backend Mein Command Execute Kab Karna Padta Hai?

Kabhi-kabhi backend ko system commands run karne padte hain, jaise:

- File conversion (e.g., `ffmpeg` se video convert karna).
- System status check (e.g., `df -h` se disk space check karna).
- External tool integration.

**Problem**: Agar user input direct command mein jata hai, toh attacker malicious command inject kar sakta hai.

### 17.2 Vulnerable Code:

Listing 15: Vulnerable Command Execution

```
const express = require('express');
const { exec } = require('child_process');
const app = express();

app.get('/check-disk', (req, res) => {
  const disk = req.query.disk; // User input
  const command = `df -h ${disk}`; // Direct input command mein
  exec(command, (err, stdout, stderr) => {
    if (err) return res.status(500).json({ error: 'Kuch galat ho gaya' });
    res.send(stdout);
  });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 17.3 Kyun Vulnerable Hai?

- Agar attacker `disk` mein `; rm -rf /` daal de, toh command banega:

```
df -h ; rm -rf /
```

- Yeh disk check ke baad pura system delete kar dega.
- User input ko direct command mein daal rahe ho bina sanitize kiye.

## 17.4 Secure Code:

Listing 16: Secure Command Execution

```
const express = require('express');
const { execFile } = require('child_process');
const app = express();

app.get('/check-disk', (req, res) => {
  const disk = req.query.disk;
  // Input validation - sirf valid disk names allow karo
  if (!disk || !/^[a-zA-Z0-9_-]+$/.test(disk)) {
    return res.status(400).json({ error: 'Galat disk name, bhai!' });
  }
  // execFile use karo jo arguments alag se pass karta hai
  execFile('df', ['-h', disk], (err, stdout, stderr) => {
    if (err) return res.status(500).json({ error: 'Kuch galat ho gaya' });
    res.send(stdout);
  });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 17.5 Kyun Secure Hai?

- **Input validation**: Regex se check kiya ki `disk` mein sirf alphanumeric aur `_ -` ho, no special characters like `;`.
- **execFile**: Yeh command string ke bajaye arguments array ke roop mein leta hai, toh injection ka risk nahi hai.
- Attacker ab `; rm -rf /` nahi daal sakta.

## 18 Example 3: XML Injection

### 18.1 Vulnerable Code:

Listing 17: Vulnerable XML Parsing

```
const express = require('express');
const xml2js = require('xml2js');
const app = express();

app.post('/parse-xml', (req, res) => {
  const xmlData = req.body.xml; // User input
  xml2js.parseString(xmlData, (err, result) => {
    if (err) return res.status(500).json({ error: 'XML galat hai' });
    res.json(result);
  });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 18.2 Kyun Vulnerable Hai?

- Agar attacker XML mein malicious entities daal de jaise:

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<root>&xxe;</root>
```

- Yeh **XXE (XML External Entity)** attack banega aur sensitive files (jaise `/etc/passwd`) leak ho sakte hain.

### 18.3 Secure Code:

Listing 18: Secure XML Parsing

```
const express = require('express');
const xml2js = require('xml2js');
const app = express();
```

```
4
5  app.post('/parse-xml', (req, res) => {
6    const xmlData = req.body.xml;
7    const parser = new xml2js.Parser({ noent: false }); // External
         entities disable karo
8    parser.parseString(xmlData, (err, result) => {
9      if (err) return res.status(500).json({ error: 'XML galat hai' });
10     res.json(result);
11   });
12 });
13
14 app.listen(3000, () => console.log('Server chal raha hai'));
```

### 18.4 Kyun Secure Hai?

- `noent: false` se external entities processing disable ho gaya, toh XXE attack nahi hoga.

## 19 Injection Preventing Guide

- **Validate & Sanitize Inputs:**
  - Har user input ko check karo ki woh expected format mein hai ya nahi.
  - **Regex**: Pattern use karo (e.g., `/^[a-zA-Z0-9]+$/`) taaki special characters reject ho jayein.
- **Parameterized Queries:**
  - SQL queries mein `?` placeholders use karo (upar SQL example dekho).
  - Yeh input ko code ke roop mein execute hone se rokta hai.
- **ORM Libraries:**
  - Sequelize ya Mongoose jaise ORM use karo jo automatically queries ko sanitize karte hain.
  - Example: `User.findOne({ where: { username } })` - yeh injection safe hai.
- **Escape & Encode Output:**
  - Output ko webpage pe dikhane se pehle encode karo taaki `<script>` jaise tags execute na ho.
  - Special characters (`<`, `>`, `&`) ko harmless banaya jata hai:
    - `<` → `<`
    - `>` → `>`
    - `&` → `&`

  - Yeh XSS (Cross-Site Scripting) rokta hai.

- **File Upload Security Checks:**
  - **File Extension Check**: Sirf allowed extensions (e.g., `.jpg`, `.pdf`) accept karo.
    * Kyun? Attacker `.php` ya `.exe` upload kar sakta hai jo server pe execute ho.
  - **MIME Type Check**: File ka actual type check karo (e.g., `image/jpeg`), kyunki extension fake ho sakti hai.
  - **Limit File Size**: Bada file upload rokne ke liye limit set karo (e.g., 5MB), taaki server crash na ho.
  - **Generate Secure Filename**: User ka filename use mat karo, random name banao (e.g., `uuidv4() + '.jpg'`) taaki overwrite ya path traversal attack na ho.

## 19.1 Example File Upload Secure Code:

Listing 19: Secure File Upload

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const { v4: uuidv4 } = require('uuid');
const app = express();

const upload = multer({
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB limit
  fileFilter: (req, file, cb) => {
    const allowedTypes = ['image/jpeg', 'image/png', 'application/pdf'];
    if (!allowedTypes.includes(file.mimetype)) {
      return cb(new Error('Sirf images ya PDF allowed!'));
    }
    cb(null, true);
  },
  storage: multer.diskStorage({
    destination: 'uploads/',
    filename: (req, file, cb) => {
      const ext = path.extname(file.originalname);
      cb(null, `${uuidv4()}${ext}`); // Random secure filename
    }
  })
});

app.post('/upload', upload.single('file'), (req, res) => {
  res.json({ message: 'File uploaded successfully', filename: req.file.filename });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

### 19.2 Kyun Yeh Secure Hai?

- File size limited hai, MIME type check hota hai, aur random filename generate hota hai, toh attacker malicious file upload nahi kar sakta.

---

# Insecure Design in Web Applications

**Insecure Design in Web Applications** Bug Bounty Hunter March 29, 2025

## 20 Insecure Design Kya Hai?

Insecure Design tab hota hai jab tumhara application ka design hi galat hota hai, matlab security ko pehle se socha hi nahi gaya. Yeh coding mistake se alag hai—yeh problem planning aur architecture level pe hoti hai. Agar design mein hi security weak hai, toh chahe code kitna bhi acha likho, attacker usko tod sakta hai.

**Simple Example:**

- Ek website banaayi jisme admin aur user dono same login page use karte hain, aur koi role-based check nahi hai design mein.
- Ya ek system jisme sensitive data (jaise passwords) plain text mein store karne ka plan hai.

Insecure design ka matlab hai ki tumne threat modeling nahi kiya, ya security requirements ko ignore kar diya.

## 21 Insecure Design Ka Ek Chhota Example

Maan lo ek e-commerce app bana rahe ho. Tumne design yeh socha ki har user apne order ka status check kar sakta hai bas order ID daal ke. Lekin design mein yeh nahi socha ki koi dusre user ka order ID guess kar ke uska data dekh sakta hai. Yeh insecure design hai kyunki tumne access control ko pehle se plan nahi kiya.

## 22 Vulnerable Code vs Secure Code Example

## 22.1 Vulnerable Code: Order Status Check

Listing 20: Vulnerable Order Status Check

```
const express = require('express');
const app = express();

const orders = [
  { id: 1, userId: 101, status: 'Shipped' },
  { id: 2, userId: 102, status: 'Pending' }
];

app.get('/order/status', (req, res) => {
  const orderId = parseInt(req.query.orderId); // User input se order ID
  const order = orders.find(o => o.id === orderId);
  if (order) {
    res.json({ order });
  } else {
    res.status(404).json({ error: 'Order nahi mila' });
  }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 22.2 Kyun Vulnerable Hai?

- Yeh design galat hai kyunki koi bhi user kisi bhi `orderId` daal ke dusre user ka order dekh sakta hai.
- Koi authentication ya authorization check nahi hai ki yeh order current user ka hai ya nahi.
- **Attacker** bas `orderId=1`, `orderId=2` try kar ke sabka data le sakta hai.
- Yeh **Broken Access Control** ka bhi example ban jata hai, lekin yeh problem design level pe hai—tumne pehle se yeh socha hi nahi ki user-specific access chahiye.

## 22.3 Secure Code: Order Status Check

Listing 21: Secure Order Status Check

```
const express = require('express');
const app = express();

// Dummy function for logged-in user
const getCurrentUser = (req) => {
  return { id: 101, name: 'Test User' }; // Real mein session ya token se aayega
};

const orders = [
```

```
  { id: 1, userId: 101, status: 'Shipped' },
  { id: 2, userId: 102, status: 'Pending' }
];

app.get('/order/status', (req, res) => {
  const currentUser = getCurrentUser(req); // Current user ka data
  const orderId = parseInt(req.query.orderId);
  const order = orders.find(o => o.id === orderId && o.userId ===
      currentUser.id); // User-specific check
  if (order) {
    res.json({ order });
  } else {
    res.status(404).json({ error: 'Order nahi mila ya tera nahi hai,
       bhai!' });
  }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

## 22.4 Kyun Secure Hai?

- Ab design mein yeh socha gaya hai ki har user sirf apne orders hi dekh sakta hai.
- `o.userId === currentUser.id` check karta hai ki order current logged-in user ka hai ya nahi.
- **Attacker** ab dusre user ka `orderId` daal ke data nahi dekh sakta.
- Yeh secure design ka example hai kyunki access control pehle se plan kiya gaya.

# 23 Insecure Design Ke Aur Examples

- **No Rate Limiting in Design:**
  - Agar tumne login page ke design mein rate limiting (e.g., 5 attempts per IP) nahi socha, toh attacker brute force kar sakta hai. Yeh coding se pehle planning ka issue hai.
- **Hard-Coded Sensitive Logic:**
  - Agar design mein yeh decide kiya ki API keys ya passwords code mein hi rahenge (jaise `const API_KEY = 'xyz'`), toh yeh insecure design hai. Keys ko external vault mein rakhna chahiye tha.
- **Missing Threat Modeling:**
  - Agar tumne pehle se nahi socha ki attacker file upload ke through malicious files daal sakta hai, aur design mein validation nahi rakha, toh yeh insecure hai.

# 24 Insecure Design Kyun Hota Hai?

- **Jaldi Baazi**: Developers ya designers security ko baad mein sochenge bol ke skip kar dete hain.
- **Knowledge Ki Kami**: Team ko pata hi nahi hota ki threat modeling kya hota hai.
- **Wrong Assumptions**: "Yeh feature toh sirf internal use ke liye hai" soch ke security ignore kar dete hain.

# 25 Insecure Design Ko Kaise Rokna Hai?

- **Threat Modeling Karo:**
  - Pehle socho ki attacker kahan se attack kar sakta hai. Har feature ke liye risks likho.
  - Example: "Order status check mein koi bhi order ID daal sakta hai" → isko fix karo.
- **Security Requirements Define Karo:**
  - Design phase mein hi decide karo ki authentication, authorization, encryption kahan-kahan lagega.
  - Example: "Har user sirf apna data dekh sakta hai" ko requirement banao.
- **Secure by Default:**
  - Default settings secure rakho. Jaise, agar koi feature optional hai, toh usko off rakho jab tak zarurat na ho.
- **Review Design:**
  - Code likhne se pehle design ko security expert se review karwao taaki galtiyaan pehle pakdi ja sakein.

# 26 Real-World Example in Hinglish

Maan lo ek banking app bana rahe ho. Tumne design yeh socha ki `/transfer-money` endpoint banao jisme `amount` aur `toAccount` user input se aayega. Lekin yeh nahi socha ki koi user apne account se kisi aur ke account mein paisa transfer kar sakta hai bina check ke.

### 26.1 Vulnerable Design Output:

- User A (ID: 101) `/transfer-money?amount=500&toAccount=102` hit karta hai.
- Paisa User B (ID: 102) ke account mein chala jata hai bina yeh check kiye ki User A ka yeh right hai ya nahi.

### 26.2 Secure Design Output:

- Design mein yeh add kiya ki `/transfer-money` pe current user ka ID check hoga aur sirf uske linked accounts mein transfer allowed hoga.
- Ab User A sirf apne accounts mein hi paisa bhej sakta hai.

## 27 Conclusion

Insecure Design ka matlab hai ki tumne building banane se pehle foundation hi galat rakha. Chahe code kitna bhi acha ho, agar design mein security nahi sochi gayi, toh attacker usko tod dega. Upar wale examples se tujhe idea mil gaya hoga ki kaise design level pe sochna hai. Yeh tere notes ke liye perfect hai, aur job mein bhi kaam aayega jab tu code review karega ya vulnerabilities dhoondhega.

==========

# Secure Coding Guide: Security Misconfiguration aur Hardening

## 28 Introduction

Alright, bhai! Let's dive into Section 6: **Secure Coding** with a focus on **Security Misconfiguration** and **Security Hardening** in Hinglish. Main tujhe har cheez step-by-step samjhaunga, notes ke style mein, aur code ke saath comments bhi daalunga taaki tu apne job mein bug bounty hunting ya code review mein pro ban jaye. Chalo shuru karte hain!

## 29 Secure Coding - Security Misconfiguration aur Security Hardening

### 29.1 Security Misconfiguration Kya Hai?

Security misconfiguration hota hai jab tumhara application ya server sahi tarike se configure nahi hota, aur is wajah se attacker usme ghus sakta hai. Yeh tab hota hai jab **default settings, unnecessary features, ya galat configurations** chhod di jati hain. Iska matlab hai ki tumne apne fort ke darwaze khulle chhod diye!

### 29.2 Security Hardening Kya Hai?

Security hardening ka matlab hai apne application ya server ko itna mazboot banana ki attacker ke liye ghusna mushkil ho jaye. Isme **default cheezein hatao, secure settings lagao**, aur latest security features use karo.

## 30 Common Security Misconfiguration Issues aur Unko Kaise Fix Karna Hai

### 30.1 Default Accounts aur Passwords

- **Problem**: Bahut saare applications default accounts (jaise admin:admin ya root:password) ke saath aate hain. Agar yeh change nahi kiye, toh attacker guess kar lega.
- **Fix**: Default accounts disable karo ya passwords strong banao.
- Example: Agar tum ek app bana rahe ho, toh production mein jaane se pehle test users (jaise test:test123) hata do.

### 30.2 Default Pages, Ports, Services

- **Problem**: Kai baar default pages (jaise WordPress ka license.txt ya readme.html) ya services (jaise HTTP port 80 bina TLS ke) live chhod di jati hain, jo attacker ko info deti hain.
- **Fix**: Inko hatao ya disable karo.
- Example: WordPress ke default files (license.txt, readme.html) delete kar do taaki attacker ko version pata na chale.

### 30.3 Default Headers

- **Problem**: Server headers (jaise "Server: Apache/2.4.41") version info dete hain, jo attacker exploit ke liye use kar sakta hai.

- **Fix**: Headers hatao ya customize karo.

## 30.4 Debug Mode

- **Problem**: Agar debug mode on hai (jaise Django mein DEBUG = True), toh error pages pe sensitive info (jaise DB details) leak ho sakta hai.
- **Fix**: Production mein debug mode off rakho.
- Example: Django mein ‘settings.py‘ mein ‘DEBUG = False‘ set karo.

## 30.5 Error Handling aur Stack Traces

- **Problem**: Agar error messages mein stack traces ya sensitive info (jaise ”SQL syntax error at line 42”) dikhayi dete hain, toh attacker ko attack plan banane mein madad milti hai.
- **Fix**: Generic error messages dikhao aur sensitive info expose mat karo.
- Example: ”SQL syntax error” ke bajaye ”Kuch galat ho gaya, baad mein koshish karo” dikhao.

## 30.6 Code aur Sensitive Data Exposure

- **Problem**: Agar error messages ya logs mein passwords, API keys, ya DB details leak ho jayein, toh attacker unko use kar sakta hai.
- **Fix**: Logs mein bhi sensitive data mat daalo, aur custom error pages banao.

# 31 Secure Coding Examples in Express.js

## 31.1 Example 1: Debug Mode aur Error Handling

Vulnerable Code:

Listing 22: Vulnerable Debug Mode

```
const express = require('express');
const app = express();

// Vulnerable Code - Debug mode on aur detailed errors
app.get('/test', (req, res) => {
    throw new Error('Kuch toh galat hai!'); // Error throw kar rahe
        hain
});
```

```
8
9  // Yeh middleware har error ko catch karta hai aur detailed stack trace
       dikhata hai
10 app.use((err, req, res, next) => {
11     res.status(500).send(err.stack); // Attacker ko pura stack trace
           mil jayega
12 });
13
14 app.listen(3000, () => console.log('Server chal raha hai'));
```

- **Kyun Vulnerable Hai?**: Yeh code error ka pura stack trace (line numbers, file paths) dikhata hai, jo attacker ko system ke bare mein info deta hai.

Secure Code:

Listing 23: Secure Debug Mode

```
1  // Secure Code - Generic error message aur debug off
2  const express = require('express');
3  const app = express();
4
5  // Production mein debug mode off hai, toh environment check karo
6  const isProduction = process.env.NODE_ENV === 'production';
7
8  app.get('/test', (req, res) => {
9      throw new Error('Kuch toh galat hai!'); // Error throw kar rahe
           hain
10 });
11
12 // Custom error handling middleware
13 app.use((err, req, res, next) => {
14     // Agar production mein hai, toh generic message dikhao
15     if (isProduction) {
16         res.status(500).json({ error: 'Kuch galat ho gaya, baad mein
               koshish karo' });
17     } else {
18         // Development mein thoda detail dikha sakte hain
19         res.status(500).json({ error: err.message });
20     }
21 });
22
23 app.listen(3000, () => console.log('Server chal raha hai'));
```

- **Kyun Secure Hai?**: Production mein generic message dikhata hai, aur stack trace hide karta hai. Development mein thoda detail deta hai debugging ke liye.

## 31.2 Example 2: Default Headers Remove Karna

Vulnerable Code:

Listing 24: Vulnerable Headers

```
1  const express = require('express');
2  const app = express();
```

```

// Vulnerable Code - Default headers expose karte hain server info
app.get('/', (req, res) => {
    res.send('Hello, bhai!');
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

- **Kyun Vulnerable Hai?**: Yeh code "X-Powered-By: Express" header bhejta hai, jo attacker ko batata hai ki Express use ho raha hai.

Secure Code:

Listing 25: Secure Headers

```
// Secure Code - Headers remove karo
const express = require('express');
const app = express();

// Default header hatao
app.disable('x-powered-by');

app.get('/', (req, res) => {
    res.send('Hello, bhai!');
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

- **Kyun Secure Hai?**: 'app.disable('x-powered-by')' se Express ka default header hat jata hai, aur attacker ko extra info nahi milti.

## 31.3 Example 3: Custom Error Pages

Listing 26: Custom Error Pages

```
const express = require('express');
const app = express();

// 404 error ke liye custom page
app.use((req, res, next) => {
    res.status(404).json({ error: 'Bhai, yeh page nahi mila!' }); //
        404 ke liye generic message
});

// 500 error ke liye custom handler
app.use((err, req, res, next) => {
    console.error(err.stack); // Internal log mein error save karo
    res.status(500).json({ error: 'Server mein kuch gadbad hai, baad
        mein try karo' });
});

app.get('/fail', (req, res) => {
    throw new Error('Forced error'); // Test ke liye error
});

```

```
19 app.listen(3000, () => console.log('Server chal raha hai'));
```

- **Kyun Secure Hai?**: Yeh code 404 aur 500 errors ke liye custom messages dikhata hai, sensitive info hide karta hai, aur errors ko internally log karta hai.

# 32 Secure Configurations for Servers

## 32.1 IIS (Internet Information Services) Security Settings

- **Request Filtering**: Unwanted HTTP requests (jaise HEAD ya TRACE) block karo.
- **IP and Domain Restrictions**: Specific IPs ya domains ko allow/block karo.
- **Dynamic IP Restrictions**: Agar koi IP zyada requests bhejta hai, toh auto-block karo.

## 32.2 Apache Security Settings

- **ModSecurity**: Web application firewall lagao jo common attacks (SQL injection, XSS) rokta hai.
- **AllowOverride None**: '.htaccess' files disable karo taaki security settings override na ho.
- **Directory Indexing Off**: 'Options -Indexes' set karo taaki directory listing na dikhe.

## 32.3 Django Security Features

- **CSRF Protection**: Django forms mein CSRF token automatically aata hai.
- **Query Parameterization**: Django querysets SQL injection se safe hain kyunki parameterized queries use karte hain.

# 33 Production Se Pehle Checklist

1. **Default Accounts Disable Karo**: Admin/test accounts hatao ya passwords change karo.
2. **Insecure Services Off Karo**: HTTP (port 80) ke bajaye HTTPS (port 443) use karo.

3. **Default Pages Hatao**: WordPress ke readme.txt ya license.txt delete karo.

4. **Debug Mode Off Karo**: Django mein 'DEBUG = False', Express mein 'NODE$_ENV$ = *production*'.**Unnecessary Functionalities Hatao** : $Unusedports, services, yatestroutesdisablekaro.$

## 34 Error Handling Best Practices

- **Generic Messages**: Users ko "Kuch galat ho gaya" jaisa message dikhao.
- **Internal Logs**: Detailed errors (stack traces) server logs mein save karo, public se hide rakho.
- **Custom Error Pages**: 404, 500 jaise errors ke liye alag pages banao.

==========

# Secure Coding Guide: Vulnerable and Outdated Components

## 35 Introduction

Alright, bhai! Since you're shifting your work to a source code reviewer, I'll help you create a new section for your notes based on your request. Yeh section **"Vulnerable and Outdated Components"** ke baare mein hoga, jo OWASP Top 10 ka ek important part hai. Main tujhe vulnerable code, secure code, aur har line ke saath comments dunga taaki tu har cheez ko achhe se samajh sake aur apne job mein use kar sake. Sab kuch Hinglish mein hoga, jaise tune bola, aur teri uploaded notes ke style mein likha jayega. Chalo shuru karte hain!

## 36 Section 7: Secure Coding - Vulnerable and Outdated Components

### 36.1 7.1 Vulnerable and Outdated Components Kya Hai?

Bhai, yeh problem tab aati hai jab tumhare code mein purane ya outdated libraries, frameworks, ya components use ho rahe hote hain jismein already security bugs ya vulnerabilities pata chal chuki hoti hain. Agar inko update nahi kiya gaya, toh attacker inka fayda utha sakta hai aur system ko hack kar sakta hai. Yeh ek common issue hai kyunki developers aksar sochte hain **"chal raha hai toh chhod do,"** lekin yeh galat hai.

**Simple Example:**

- Maan lo tum Express.js ke version 4.16.0 use kar rahe ho, jo purana hai aur isme ek known vulnerability hai (CVE-2019-XXXX) jisse attacker Denial of Service (DoS) attack kar sakta hai.
- Agar tum isko update nahi karte, toh attacker bas ek malformed request bhejke server crash kar sakta hai.

Ab main tujhe vulnerable code aur secure code ka example dunga, aur yeh bhi bataunga ki kaise inko find aur patch karna hai.

## 36.2 7.2 Example: Outdated Express.js Version

### 36.2.1 7.2.1 Vulnerable Code

**Listing 22: Vulnerable Code with Outdated Express.js**

Listing 27: Vulnerable Code with Outdated Express.js

```
const express = require('express'); // Yeh version 4.16.0 hai jo outdated hai
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, Bhai! Server chal raha hai.');
});

app.listen(3000, () => console.log('Server chal raha hai port 3000 pe'));
```

**Line-by-Line Explanation with Comments:**

Listing 28: Vulnerable Code with Comments

```
const express = require('express'); // Yeh line Express.js ko import karti hai, lekin agar yeh version 4.16.0 hai toh isme known vulnerabilities hain jaise DoS attack ka risk
const app = express(); // Express app banaya gaya, yeh basic setup hai
app.get('/', (req, res) => { // '/' route pe GET request handle karne ka code
    res.send('Hello, Bhai! Server chal raha hai.'); // Response mein simple message bhej raha hai
});
app.listen(3000, () => console.log('Server chal raha hai port 3000 pe')); // Server 3000 port pe start ho raha hai, lekin purana version security risk la sakta hai
```

### 36.2.2 7.2.2 Kyun Vulnerable Hai?

- Yeh code Express.js ke version 4.16.0 pe chal raha hai, jo purana hai aur isme security patches nahi hain.
- Is version mein ek known issue hai (hypothetical CVE-2019-XXXX) jisse attacker malformed input bhejke server ko crash kar sakta hai.
- Agar attacker jaan bujhkar galat request bhejta hai (jaise invalid headers), toh server hang ho sakta hai ya memory leak ho sakta hai.
- Koi version check ya update ka dhyan nahi rakha gaya.

### 36.2.3 7.2.3 Secure Code

**Listing 23: Secure Code with Updated Express.js**

Listing 29: Secure Code with Updated Express.js

```
const express = require('express'); // Yeh latest version 4.18.2 ya uske upar ka hai, jo secure hai
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, Bhai! Secure server chal raha hai.');
});

app.listen(3000, () => console.log('Server chal raha hai port 3000 pe'));
```

**Line-by-Line Explanation with Comments:**

Listing 30: Secure Code with Comments

```
const express = require('express'); // Express.js ko import kiya, lekin ab latest version (jaise 4.18.2) use kar rahe hain jisme security patches hain
const app = express(); // Express app banaya, yeh same hai lekin ab secure foundation pe hai
app.get('/', (req, res) => { // '/' route pe GET request handle kar raha hai
    res.send('Hello, Bhai! Secure server chal raha hai.'); // Response mein message bhej raha hai, ab safe hai
});
app.listen(3000, () => console.log('Server chal raha hai port 3000 pe')); // Server 3000 port pe chal raha hai, aur latest version ke saath secure hai
```

### 36.2.4 7.2.4 Kyun Secure Hai?

- Latest Express.js version (jaise 4.18.2 ya uske upar) use kiya gaya, jisme purane bugs aur vulnerabilities fix ho chuke hain.

- Developer ne package.json mein version update kiya aur npm install chalaya taaki latest code mile.
- Ab attacker ke DoS attack ka chance kam hai kyunki new version mein better error handling aur security fixes hain.

## 36.3 7.3 Vulnerable Outdated Components Kaise Find Karein?

Bhai, jab tu source code review karega, toh outdated components ko dhoondhne ke liye yeh steps follow kar:

1. **Package.json Check Karo:**
   - Project ke root folder mein 'package.json' file kholo.
   - Dekho ki dependencies aur devDependencies mein kaunse packages hain aur unke versions kya hain.
- Example: Agar '"express": "$^4$.16.0"'$likhahai, tohyehpuranahai$.
2. **NPM Outdated Command Use Karo:**
   - Terminal mein jao aur 'npm outdated' run karo.
   - Yeh list dega ki kaunse packages outdated hain, unka current version kya hai, aur latest version kya hai.
   - Output aisa hoga:

```
Package  Current  Wanted  Latest
express  4.16.0   4.16.0  4.18.2
```

3. **CVE Database Check Karo:**
   - Websites jaise `[NVD (National Vulnerability Database)](https://nvd.nist.gov/)` pe jao.
   - Package name aur version daal ke check karo ki usme koi known vulnerability hai ya nahi.
   - Example: "Express 4.16.0" search karo aur CVE entries dekho.
4. **Dependency Scanners Use Karo:**
   - Tools jaise 'npm audit' ya 'Snyk' chalao.
   - 'npm audit' run karne pe yeh bolega:

```
found 3 vulnerabilities (1 low, 2 moderate)
run 'npm audit fix' to fix them
```

   - Yeh automatically vulnerabilities dhoondh lega.
5. **Code Mein Manual Check:**

- Agar koi package directly import kiya gaya hai (jaise ‘require('express')‘), toh uska version package.json se match karo aur check karo ki woh latest hai ya nahi.

## 36.4 7.4 Vulnerable Components Ko Kaise Patch Karein?

1. **Version Update Karo:**
   - ‘package.json‘ mein version change karo, jaise:

     ```
     "express": "^4.18.2"
     ```

   - Phir ‘npm install‘ run karo taaki latest version install ho jaye.
2. **NPM Audit Fix Use Karo:**
   - ‘npm audit fix‘ run karo, yeh automatically chhoti vulnerabilities ko patch kar dega.
   - Agar breaking changes hain, toh manually update karna padega.
3. **Breaking Changes Check Karo:**
   - Latest version ke release notes padho (GitHub pe milenge).
   - Dekho ki koi bada change hai ya nahi, taaki code toot na jaye.
4. **Test Karo:**
   - Update ke baad pura code test karo taaki pata chale ki koi functionality break toh nahi hui.
5. **Auto-Update Tools:**
   - ‘Dependabot‘ (GitHub pe) ya ‘Renovate‘ use karo jo automatically PR banayega jab koi package outdated ho.

## 36.5 7.5 Real-World Example in Hinglish

Maan lo ek project mein tu ‘lodash‘ library use kar raha hai version 4.17.15 pe. Yeh version purana hai aur isme ek vulnerability hai (CVE-2021-23337) jisse attacker malicious input bhejke code execute kar sakta hai.

**Vulnerable Setup:**

- ‘package.json‘ mein: ‘”lodash”: ”$^4$.17.15”‘$Code$ : ‘$const\_require('lodash');$‘

- Attacker input bhejta hai aur server crash ho jata hai.

**Secure Fix:**

- ‘package.json‘ update karo: “”lodash”: ”$^{4}$.17.21” “*npminstall‘chalao*.

- Ab vulnerability fix ho gayi, aur attacker ka attack kaam nahi karega.

### 36.6 7.6 Prevention Tips

- **Regular Updates:** Har mahine ‘npm outdated‘ check karo aur packages update karo.
- **Minimal Dependencies:** Sirf wahi packages use karo jo zaruri hain, zyada packages matlab zyada risk.
- **Security Alerts:** GitHub pe Dependabot enable karo taaki alert mile.
- **Documentation Padho:** Naye version ke changes samajh lo pehle.

==========

# Secure Coding Guide: Identification and Authentication Failures

**Secure Coding Guide: Identification and Authentication Failures** Your Name
March 28, 2025

## 37 Introduction

Alright, bhai! Ab hum ek naye section banayenge tere notes ke liye: **”Secure Coding - Identification and Authentication Failures”**. Yeh OWASP Top 10 ka ek important part hai aur isme main tujhe sab kuch Hinglish mein explain karunga—weak authentication policies, brute force aur credential stuffing attacks, weak passwords, ineffective credential recovery, lack of MFA (Multi-Factor Authentication), weak session management, aur jo bhi extra points baki hain, unko bhi add kar dunga. Har cheez ke liye vulnerable code, secure code, aur line-by-line comments dunga taaki tu apne source code review ke kaam mein pro ban jaye. Chalo shuru karte hain!

## 38 Section 8: Secure Coding - Identification and Authentication Failures

## 38.1 8.1 Identification and Authentication Failures Kya Hai?

Bhai, yeh problem tab hoti hai jab system mein user ko identify aur authenticate karne ka process weak ya galat hota hai. Matlab, agar login system mein kami hai, toh attacker asani se account takeover kar sakta hai. Isme **weak passwords, no MFA, weak session management, aur galat forgot-password process** jaise issues aate hain. Agar yeh theek nahi kiya, toh brute force, credential stuffing, ya session hijacking jaise attacks ho sakte hain.

**Common Issues:**

- Weak Authentication Policies (kamzor login checks).
- Vulnerable to Brute Force aur Credential Stuffing Attacks.
- Weak Password Usage (simple passwords jaise "123456").
- Weak Credential Recovery/Forgot-Password Process.
- Lack of Multi-Factor Authentication (MFA).
- Weak Session Process (session ID galat tarike se use hona).
- Exposing Session ID in URL, Reusing Session ID, Not Invalidating Session After Logout.

Ab ek-ek karke examples ke saath samjhaunga.

## 38.2 8.2 Example 1: Weak Authentication (Vulnerable to Brute Force)

### 38.2.1 8.2.1 Vulnerable Code

**Listing 24: Vulnerable Login with No Brute Force Protection**

Listing 31: Vulnerable Login with No Brute Force Protection

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    // Dummy check, real mein database se hoga
    if (username === 'admin' && password === 'pass123') {
        res.json({ message: 'Login successful, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat credentials' });
    }
});
```

```

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 32: Vulnerable Login with Comments

```
const express = require('express'); // Express import kiya, basic setup hai
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye middleware
app.post('/login', (req, res) => { // '/login' endpoint pe POST request handle kar raha hai
    const { username, password } = req.body; // Request body se username aur password liya
    if (username === 'admin' && password === 'pass123') { // Direct check, koi rate limit ya protection nahi
        res.json({ message: 'Login successful, bhai!' }); // Agar match hua toh success message
    } else {
        res.status(401).json({ error: 'Galat credentials' }); // Nahi toh error
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 pe chal raha hai
```

### 38.2.2 8.2.2 Kyun Vulnerable Hai?

- Koi rate limiting nahi hai, toh attacker infinite login attempts bhej sakta hai (brute force attack).
- Weak password ("pass123") use kiya gaya, jo guess karna aasan hai.
- Credential stuffing ke liye bhi vulnerable hai kyunki koi extra check (jaise CAPTCHA) nahi hai.
- Agar attacker ek leaked password list se try kare, toh yeh asani se crack ho sakta hai.

### 38.2.3 8.2.3 Secure Code

**Listing 25: Secure Login with Rate Limiting and Strong Password Check**

Listing 33: Secure Login with Rate Limiting

```
const express = require('express');
const rateLimit = require('express-rate-limit'); // Rate limiting ke liye
const app = express();
app.use(express.json());

```

```
// Rate limit: 5 attempts per IP in 15 minutes
const loginLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minute window
    max: 5, // 5 requests allowed
    message: 'Bhai, zyada try kar raha hai, 15 minute ruk!'
});

app.post('/login', loginLimiter, (req, res) => {
    const { username, password } = req.body;
    // Password strength check (min 8 chars, 1 number, 1 special char)
    const passwordRegex = /^(?=.*[0-9])(?=.*[!@#$%^&*])[a-zA-Z0-9!@#$%^&*]{8,}$/;
    if (!passwordRegex.test(password)) {
        return res.status(400).json({ error: 'Password kamzor hai, bhai!' });
    }
    if (username === 'admin' && password === 'Secure@123') {
        res.json({ message: 'Login successful, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat credentials' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 34: Secure Login with Comments

```
const express = require('express'); // Express import kiya
const rateLimit = require('express-rate-limit'); // Rate limiting package import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
const loginLimiter = rateLimit({ // Rate limiter configure kiya
    windowMs: 15 * 60 * 1000, // 15 minute ka time window
    max: 5, // Sirf 5 attempts allow honge
    message: 'Bhai, zyada try kar raha hai, 15 minute ruk!' // Custom message jab limit cross ho
});
app.post('/login', loginLimiter, (req, res) => { // Login endpoint pe rate limiter add kiya
    const { username, password } = req.body; // Username aur password liya
    const passwordRegex = /^(?=.*[0-9])(?=.*[!@#$%^&*])[a-zA-Z0-9!@#$%^&*]{8,}$/; // Strong password ka regex (8+ chars, number, special char)
    if (!passwordRegex.test(password)) { // Password check kiya
        return res.status(400).json({ error: 'Password kamzor hai, bhai!' }); // Agar weak hai toh error
    }
    if (username === 'admin' && password === 'Secure@123') { // Strong password ke saath check
        res.json({ message: 'Login successful, bhai!' }); // Success message
    } else {
        res.status(401).json({ error: 'Galat credentials' }); // Error message
```

```
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.2.4 8.2.4 Kyun Secure Hai?

- 'express-rate-limit' use kiya taaki brute force attack ruk jaye (5 attempts ke baad block).
- Password strength check add kiya (regex) taaki weak passwords reject ho jayein.
- Strong password ("Secure@123") example diya, jo guess karna mushkil hai.
- Credential stuffing ke liye bhi protection hai kyunki attempts limited hain.

## 38.3 8.3 Example 2: Weak Credential Recovery/Forgot-Password Process

### 38.3.1 8.3.1 Vulnerable Code

**Listing 26: Vulnerable Forgot-Password Process**

Listing 35: Vulnerable Forgot-Password Process

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/forgot-password', (req, res) => {
    const { email } = req.body;
    // Dummy user check
    if (email === 'test@example.com') {
        const resetLink = `http://example.com/reset?email=${email}`; // Email URL mein expose ho raha hai
        res.json({ message: 'Reset link bhej diya: ' + resetLink });
    } else {
        res.status(404).json({ error: 'Email nahi mila' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 36: Vulnerable Forgot-Password with Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
```

```
app.post('/forgot-password', (req, res) => { // Forgot-password endpoint
    const { email } = req.body; // Email liya request se
    if (email === 'test@example.com') { // Dummy check, real mein DB se hoga
        const resetLink = `http://example.com/reset?email=${email}`; // Reset link banaya, email expose ho raha hai URL mein
        res.json({ message: 'Reset link bhej diya: ' + resetLink }); // Link response mein bhej diya, jo galat hai
    } else {
        res.status(404).json({ error: 'Email nahi mila' }); // Agar email nahi mila toh error
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.3.2 8.3.2 Kyun Vulnerable Hai?

- Reset link mein email direct URL mein expose ho raha hai, jo attacker dekh sakta hai agar response intercept ho.
- Koi unique token nahi hai, toh attacker email guess karke reset kar sakta hai.
- Koi expiry time nahi hai link ka, toh kabhi bhi use ho sakta hai.
- Response mein reset link bhejna bhi galat hai, yeh email pe bhejna chahiye.

### 38.3.3 8.3.3 Secure Code

**Listing 27: Secure Forgot-Password Process**

Listing 37: Secure Forgot-Password Process

```
const express = require('express');
const crypto = require('crypto');
const app = express();
app.use(express.json());

app.post('/forgot-password', (req, res) => {
    const { email } = req.body;
    if (email === 'test@example.com') {
        const resetToken = crypto.randomBytes(32).toString('hex'); // Random secure token
        const resetLink = `http://example.com/reset?token=${resetToken}`; // Token ke saath link
        // Real mein yeh email pe bhejna chahiye, DB mein store karo with expiry
        console.log(`Reset link (email pe bhejo): ${resetLink}`);
        res.json({ message: 'Reset link email pe bhej diya, check karo!' });
    } else {
        res.status(404).json({ error: 'Email nahi mila' });
```

```
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 38: Secure Forgot-Password with Comments

```
const express = require('express'); // Express import kiya
const crypto = require('crypto'); // Crypto module random token ke liye
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.post('/forgot-password', (req, res) => { // Forgot-password endpoint
    const { email } = req.body; // Email liya
    if (email === 'test@example.com') { // Dummy check
        const resetToken = crypto.randomBytes(32).toString('hex'); // 32-byte random token banaya, secure hai
        const resetLink = `http://example.com/reset?token=${resetToken}`; // Token ke saath reset link banaya
        console.log(`Reset link (email pe bhejo): ${resetLink}`); // Link console pe dikha raha, real mein email pe jayega
        res.json({ message: 'Reset link email pe bhej diya, check karo!' }); // User ko generic message
    } else {
        res.status(404).json({ error: 'Email nahi mila' }); // Email nahi mila toh error
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.3.4 8.3.4 Kyun Secure Hai?

- Random token (`crypto.randomBytes`) use kiya jo guess karna mushkil hai.
- Email URL mein expose nahi ho raha, sirf token hai.
- Reset link response mein nahi bheja, user ko email check karne ko bola (real mein email bhejna chahiye).
- Token ko DB mein store karo aur 1 ghante ka expiry time do taaki secure rahe.

## 38.4 8.4 Example 3: Lack of Multi-Factor Authentication (MFA)

### 38.4.1 8.4.1 Vulnerable Code

**Listing 28: Login Without MFA**

Listing 39: Login Without MFA

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    if (username === 'admin' && password === 'pass123') {
        res.json({ message: 'Login successful, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat credentials' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 40: Login Without MFA with Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.post('/login', (req, res) => { // Login endpoint
    const { username, password } = req.body; // Username aur password liya
    if (username === 'admin' && password === 'pass123') { // Sirf password check, MFA nahi
        res.json({ message: 'Login successful, bhai!' }); // Success message
    } else {
        res.status(401).json({ error: 'Galat credentials' }); // Error message
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.4.2 8.4.2 Kyun Vulnerable Hai?

- Sirf username aur password pe depend hai, agar password leak ho gaya toh attacker direct login kar lega.
- MFA nahi hai, toh ek layer ki security missing hai.
- Agar attacker phishing se password chura le, toh koi rok nahi hai.

### 38.4.3 8.4.3 Secure Code

**Listing 29: Login with MFA (OTP Simulation)**

Listing 41: Login with MFA

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    if (username === 'admin' && password === 'Secure@123') {
        // MFA step: OTP bhejna (dummy simulation)
        const otp = Math.floor(100000 + Math.random() * 900000); // 6-digit OTP
        console.log(`OTP bheja (real mein SMS/email pe): ${otp}`);
        res.json({ message: 'OTP bhej diya, enter karo!', step: 'otp' });
    } else {
        res.status(401).json({ error: 'Galat credentials' });
    }
});

app.post('/verify-otp', (req, res) => {
    const { otp } = req.body;
    if (otp === '123456') { // Dummy OTP check, real mein DB se verify karo
        res.json({ message: 'Login successful with MFA!' });
    } else {
        res.status(401).json({ error: 'Galat OTP' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 42: Login with MFA with Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.post('/login', (req, res) => { // Login endpoint
    const { username, password } = req.body; // Username aur password liya
    if (username === 'admin' && password === 'Secure@123') { // Strong password check
        const otp = Math.floor(100000 + Math.random() * 900000); // Random 6-digit OTP banaya
        console.log(`OTP bheja (real mein SMS/email pe): ${otp}`); // OTP console pe, real mein SMS/email pe jayega
        res.json({ message: 'OTP bhej diya, enter karo!', step: 'otp' }); // User ko OTP enter karne ko bola
    } else {
        res.status(401).json({ error: 'Galat credentials' }); // Galat credentials pe error
    }
});
app.post('/verify-otp', (req, res) => { // OTP verify endpoint
    const { otp } = req.body; // OTP liya
    if (otp === '123456') { // Dummy check, real mein DB se match karo
```

```
        res.json({ message: 'Login successful with MFA!' }); // Success message
    } else {
        res.status(401).json({ error: 'Galat OTP' }); // Galat OTP pe error
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.4.4 8.4.4 Kyun Secure Hai?

- MFA add kiya gaya (OTP), toh sirf password se login nahi hoga.
- Attacker ko password ke saath OTP bhi chahiye, jo mushkil hai.
- Real mein OTP ko SMS/email pe bhejna aur DB mein store karna chahiye with expiry (e.g., 5 minutes).

## 38.5 8.5 Example 4: Weak Session Management

### 38.5.1 8.5.1 Vulnerable Code

**Listing 30: Vulnerable Session Management**

Listing 43: Vulnerable Session Management

```
const express = require('express');
const app = express();

app.get('/login', (req, res) => {
    const sessionId = '12345'; // Fixed session ID, galat hai
    res.redirect(`/dashboard?session=${sessionId}`); // Session ID URL mein expose ho raha hai
});

app.get('/dashboard', (req, res) => {
    const sessionId = req.query.session;
    if (sessionId === '12345') {
        res.send('Welcome to Dashboard, bhai!');
    } else {
        res.status(403).json({ error: 'Invalid session' });
    }
});

app.get('/logout', (req, res) => {
    res.send('Logged out, bhai!'); // Session invalidate nahi kiya
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 44: Vulnerable Session Management with Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.get('/login', (req, res) => { // Login endpoint (dummy)
    const sessionId = '12345'; // Fixed session ID, yeh predictable aur galat hai
    res.redirect(`/dashboard?session=${sessionId}`); // Session ID URL mein daal diya, expose ho raha hai
});
app.get('/dashboard', (req, res) => { // Dashboard endpoint
    const sessionId = req.query.session; // URL se session ID liya
    if (sessionId === '12345') { // Fixed ID check, koi real session management nahi
        res.send('Welcome to Dashboard, bhai!'); // Success message
    } else {
        res.status(403).json({ error: 'Invalid session' }); // Error message
    }
});
app.get('/logout', (req, res) => { // Logout endpoint
    res.send('Logged out, bhai!'); // Session invalidate nahi kiya, purana ID reuse ho sakta hai
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 38.5.2 8.5.2 Kyun Vulnerable Hai?

- Session ID fixed hai ("12345"), jo predictable hai aur attacker guess kar sakta hai.
- Session ID URL mein expose ho raha hai (?session=12345), jo logs ya browser history mein leak ho sakta hai.
- Login ke baad naya session ID nahi banaya, reuse ho raha hai.
- Logout ke baad session invalidate nahi kiya, toh purana ID kaam karta rahega.

### 38.5.3 8.5.3 Secure Code

**Listing 31: Secure Session Management**

Listing 45: Secure Session Management

```
const express = require('express');
const session = require('express-session'); // Session management ke liye
const app = express();

app.use(session({
    secret: 'mysecretkey', // Secure secret key
    resave: false, // Session ko unnecessarily save nahi karega
```

```
    saveUninitialized: false, // Uninitialized session save nahi hoga
    cookie: { secure: true, httpOnly: true } // Cookie secure aur HTTP-only
}));

app.get('/login', (req, res) => {
    req.session.user = { id: 1, name: 'admin' }; // Session mein user data store
    res.redirect('/dashboard');
});

app.get('/dashboard', (req, res) => {
    if (req.session.user) { // Session check
        res.send(`Welcome to Dashboard, ${req.session.user.name}!`);
    } else {
        res.status(403).json({ error: 'Login kar pehle, bhai!' });
    }
});

app.get('/logout', (req, res) => {
    req.session.destroy(() => { // Session invalidate kiya
        res.send('Logged out successfully, bhai!');
    });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 46: Secure Session Management with Comments

```
const express = require('express'); // Express import kiya
const session = require('express-session'); // Session management package
const app = express(); // Express app banaya
app.use(session({ // Session middleware configure kiya
    secret: 'mysecretkey', // Secret key session sign karne ke liye, real mein strong rakhna
    resave: false, // Session ko baar-baar save nahi karega
    saveUninitialized: false, // Blank session save nahi hoga
    cookie: { secure: true, httpOnly: true } // Cookie ko secure (HTTPS) aur HTTP-only banaya taaki JS se access na ho
}));
app.get('/login', (req, res) => { // Login endpoint
    req.session.user = { id: 1, name: 'admin' }; // Session mein user data store kiya
    res.redirect('/dashboard'); // Dashboard pe redirect, session ID URL mein nahi hai
});
app.get('/dashboard', (req, res) => { // Dashboard endpoint
    if (req.session.user) { // Session check kiya ki user logged in hai ya nahi
        res.send(`Welcome to Dashboard, ${req.session.user.name}!`); // User-specific message
    } else {
        res.status(403).json({ error: 'Login kar pehle, bhai!' }); // Agar session nahi toh error
    }
```

```
});
app.get('/logout', (req, res) => { // Logout endpoint
    req.session.destroy(() => { // Session ko destroy kiya taaki
        invalidate ho jaye
        res.send('Logged out successfully, bhai!'); // Success message
    });
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
    start
```

### 38.5.4 8.5.4 Kyun Secure Hai?

- 'express-session' use kiya jo secure session ID generate karta hai (random aur unpredictable).
- Session ID URL mein expose nahi hota, cookie mein store hota hai.
- Cookie 'secure' aur 'httpOnly' hai, toh HTTPS pe chalta hai aur JS se access nahi ho sakta.
- Logout pe session destroy hota hai, toh purana ID kaam nahi karega.

## 38.6 8.6 Prevention Guide for Identification and Authentication Failures

1. **Strong Authentication Policies:**
   - Password minimum 8 chars, numbers, aur special chars ke saath rakho (regex check karo).
   - Rate limiting lagao ('express-rate-limit') taaki brute force na ho.
2. **Brute Force aur Credential Stuffing Rokne Ke Liye:**
   - CAPTCHA add karo (Google reCAPTCHA) sensitive endpoints pe.
   - Account lockout feature dalo (5 galat attempts ke baad 15 minute lock).
3. **Weak Passwords Avoid Karo:**
   - Bcrypt jaise strong hashing use karo passwords ke liye (salt ke saath).
   - Common passwords (jaise "password123") ko blacklist karo.
4. **Secure Credential Recovery:**
   - Random token use karo reset links ke liye ('crypto.randomBytes').
   - Token ko DB mein store karo with expiry (e.g., 1 hour).
   - Link email pe bhejo, response mein mat dikhao.
5. **Multi-Factor Authentication (MFA):**

- OTP (SMS/email) ya TOTP (Google Authenticator) add karo.
- Har sensitive action pe MFA mandatory karo (jaise money transfer).

6. **Strong Session Management:**
   - ‘express-session‘ ya JWT use karo secure session IDs ke liye.
   - Session ID ko cookie mein store karo, URL mein mat daalo.
   - ‘secure‘ aur ‘httpOnly‘ flags lagao cookies pe.
   - Login ke baad naya session ID generate karo.
   - Logout pe session invalidate karo (‘req.session.destroy‘).
7. **Extra Tips:**
   - Session timeout set karo (e.g., 30 minute inactivity pe logout).
   - Session ID rotation implement karo har login ke baad.

## 38.7 8.7 Real-World Example in Hinglish

Maan lo ek banking app mein login sirf username aur password se hai. Attacker ek leaked password ”pass123” se brute force karta hai aur account hack kar leta hai kyunki koi rate limit ya MFA nahi tha. Session ID bhi URL mein tha (?session=12345), jo browser history se chura liya gaya.

**Secure Fix:**

- Rate limiting lagaya (5 attempts pe block).
- MFA add kiya (OTP email pe).
- Session ID cookie mein store kiya aur logout pe destroy kar diya.
- Ab attacker ke liye hack karna mushkil ho gaya.

---

# Secure Coding Guide: Identification and Authentication Failures Examples

**Secure Coding Guide: Identification and Authentication Failures Examples**
Your Name March 28, 2025

# 39 Section 9: Identification and Authentication Failures Examples

## 39.1 9.1 Example 1: Login Brute Force

### 39.1.1 9.1.1 Vulnerable Code

**Listing 32: Vulnerable Login Endpoint (Brute Force Possible)**

Listing 47: Vulnerable Login Endpoint

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    // Dummy check, real mein database se hoga
    if (username === 'admin' && password === 'pass123') {
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 48: Vulnerable Login with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body ko parse karne ke liye middleware
app.post('/login', (req, res) => { // '/login' endpoint pe POST request
    handle kar raha hai
    const { username, password } = req.body; // Request body se
       username aur password liya
    if (username === 'admin' && password === 'pass123') { // Direct
       check kiya, koi limit ya protection nahi
        res.json({ message: 'Login ho gaya, bhai!' }); // Agar
           credentials match kare toh success
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
           // Nahi toh error
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

### 39.1.2 9.1.2 Kyun Vulnerable Hai?

- Koi rate limiting ya lockout policy nahi hai, toh attacker infinite baar login try kar sakta hai (brute force attack).
- Weak password ("pass123") use kiya gaya, jo asani se guess ho sakta hai.
- Agar attacker ek script chalaye aur 1000 passwords try kare, toh yeh system usko rok nahi sakega.
- Koi log ya monitoring bhi nahi hai jo failed attempts track kare.

### 39.1.3 9.1.3 Secure Code

**Listing 33: Secure Login with Lockout Policy**

Listing 49: Secure Login with Lockout Policy

```
const express = require('express');
const app = express();
app.use(express.json());

// Dummy storage for failed attempts (real mein DB use karo)
const failedAttempts = {};

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    const ip = req.ip; // User ka IP address liya
    failedAttempts[ip] = failedAttempts[ip] || { count: 0, lockedUntil:
        null };

    // Check if IP is locked
    if (failedAttempts[ip].lockedUntil && failedAttempts[ip].
       lockedUntil > Date.now()) {
        return res.status(429).json({ error: 'Bhai, lock ho gaya, 15
           minute ruk!' });
    }

    if (username === 'admin' && password === 'Secure@123') {
        failedAttempts[ip].count = 0; // Success pe reset karo
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        failedAttempts[ip].count += 1; // Failed attempt count badhao
        if (failedAttempts[ip].count >= 5) { // 5 attempts ke baad lock
            failedAttempts[ip].lockedUntil = Date.now() + 15 * 60 *
               1000; // 15 minute lock
            return res.status(429).json({ error: '5 galat attempts, 15
               minute ke liye lock!' });
        }
        res.status(401).json({ error: 'Galat username ya password' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 50: Secure Login with Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
const failedAttempts = {}; // Failed attempts track karne ke liye object (real mein DB use karo)
app.post('/login', (req, res) => { // Login endpoint
    const { username, password } = req.body; // Username aur password liya
    const ip = req.ip; // User ka IP address liya taaki usko track kar sakein
    failedAttempts[ip] = failedAttempts[ip] || { count: 0, lockedUntil: null }; // IP ke liye entry banayi, count aur lock time ke saath
    if (failedAttempts[ip].lockedUntil && failedAttempts[ip].lockedUntil > Date.now()) { // Check kiya ki IP lock hai ya nahi
        return res.status(429).json({ error: 'Bhai, lock ho gaya, 15 minute ruk!' }); // Agar lock hai toh error bhejo
    }
    if (username === 'admin' && password === 'Secure@123') { // Strong password ke saath check
        failedAttempts[ip].count = 0; // Success pe failed attempts reset kiya
        res.json({ message: 'Login ho gaya, bhai!' }); // Success message
    } else {
        failedAttempts[ip].count += 1; // Failed attempt ka count badhaya
        if (failedAttempts[ip].count >= 5) { // Agar 5 ya zyada attempts ho gaye
            failedAttempts[ip].lockedUntil = Date.now() + 15 * 60 * 1000; // 15 minute ka lock time set kiya
            return res.status(429).json({ error: '5 galat attempts, 15 minute ke liye lock!' }); // Lock message
        }
        res.status(401).json({ error: 'Galat username ya password' }); // Normal error agar credentials galat hain
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server start
```

### 39.1.4 9.1.4 Kyun Secure Hai?

- Lockout policy add ki gayi—5 galat attempts ke baad IP 15 minute ke liye lock ho jata hai.
- Brute force attack ruk jata hai kyunki attacker infinite tries nahi kar sakta.
- Strong password ("Secure@123") use kiya, jo guess karna mushkil hai.
- Failed attempts ko track kiya ja raha hai (real mein DB ya Redis use karo).

## 39.2 9.2 Example 2: Sensitive Data in GET Request

### 39.2.1 9.2.1 Vulnerable Code

**Listing 34: Vulnerable Login with Sensitive Data in GET**

Listing 51: Vulnerable Login with GET

```
const express = require('express');
const app = express();

app.get('/login', (req, res) => {
    const username = req.query.username; // URL se username liya
    const password = req.query.password; // URL se password liya
    if (username === 'admin' && password === 'pass123') {
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 52: Vulnerable Login with GET Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.get('/login', (req, res) => { // GET request pe '/login' endpoint
   banaya
    const username = req.query.username; // Username URL query se liya
       (?username=admin)
    const password = req.query.password; // Password URL query se liya
       (?password=pass123)
    if (username === 'admin' && password === 'pass123') { // Direct
       check kiya
        res.json({ message: 'Login ho gaya, bhai!' }); // Success
           message
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
           // Error message
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   start
```

### 39.2.2 9.2.2 Kyun Vulnerable Hai?

- Sensitive data (username aur password) GET request ke URL mein hai (e.g., '/login?username=adminpassword=pass123'), jo browser history, server logs, aur network traffic mein expose ho sakta hai.
- Attacker URL intercept karke credentials chura sakta hai.

- GET request ka use login ke liye galat hai kyunki yeh data hide nahi karta.
- Weak password ("pass123") bhi hai, jo aur bura banata hai.

### 39.2.3 9.2.3 Secure Code

**Listing 35: Secure Login with POST and Encrypted Data**

Listing 53: Secure Login with POST

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body; // POST body se data liya
    if (username === 'admin' && password === 'Secure@123') {
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 54: Secure Login with POST Comments

```
const express = require('express'); // Express import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye middleware
app.post('/login', (req, res) => { // POST request pe '/login' endpoint
    banaya
    const { username, password } = req.body; // Username aur password
       body se liya, URL mein nahi
    if (username === 'admin' && password === 'Secure@123') { // Strong
       password ke saath check
        res.json({ message: 'Login ho gaya, bhai!' }); // Success
           message
    } else {
        res.status(401).json({ error: 'Galat username ya password' });
           // Error message
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   start
```

### 39.2.4 9.2.4 Kyun Secure Hai?

- POST request use kiya, toh sensitive data (username, password) URL mein nahi, body mein jata hai.

- Data HTTPS pe encrypt hota hai (agar server HTTPS use karta hai), toh logs ya network mein expose nahi hoga.
- Strong password ("Secure@123") use kiya, jo guess karna mushkil hai.
- Attacker ke liye credentials churana ab tough hai.

## 39.3 9.3 Identification and Authentication Failures Prevention Guide

### 39.3.1 9.3.1 Brute Force Attacks Ko Rokna

- **Lockout Policy Use Karo:**
  - Agar user 5 baar galat password daale, toh uska account ya IP 15 minute ke liye lock kar do.
  - Example upar wale secure code mein hai—'failedAttempts' object IP ke hisaab se track karta hai aur 5 attempts ke baad lock karta hai.
  - Real mein yeh logic database ya Redis mein store karo taaki server restart pe data na khoye.
  - **Kyun Kaam Karta Hai?** Attacker infinite tries nahi kar sakta, aur har lock ke baad wait karna padega, jo brute force ko slow karta hai.
- **Rate Limiting Add Karo:**
  - 'express-rate-limit' package use karo taaki ek IP se limited requests hi aayein (e.g., 5 requests 15 minute mein).
  - Yeh bhi brute force ko rokta hai.
- **CAPTCHA Lagao:**
  - Google reCAPTCHA add karo taaki bots automate na kar sakein.
  - Har login attempt pe CAPTCHA solve karwana padega.

### 39.3.2 9.3.2 Sensitive Data Protection

- **POST Use Karo GET Ki Jagah:**
  - Login jaise sensitive actions ke liye hamesha POST ya PUT use karo, GET mat karo kyunki GET URL mein data daal deta hai.
  - POST body mein data encrypt hota hai HTTPS pe.
- **HTTPS Enable Karo:**
  - Server pe SSL/TLS configure karo taaki data encrypted rahe network pe.
  - Agar HTTPS nahi hai, toh attacker man-in-the-middle attack se data chura sakta hai.

### 39.3.3 9.3.3 Strong Password Policies

- Password minimum 8 characters, ek number, ek special char ke saath rakho.
- Regex check use karo jaise: ‘/(?=.*[0-9])(?=.*[!@
- Common passwords (jaise ”123456”) ko blacklist karo.

### 39.3.4 9.3.4 Multi-Factor Authentication (MFA)

- OTP (SMS/email) ya TOTP (Google Authenticator) add karo.
- Har login pe doosra factor verify karo taaki sirf password se kaam na chale.

### 39.3.5 9.3.5 Session Management

- Random session IDs generate karo (‘express-session‘ use karo).
- Session ID ko cookie mein store karo, URL mein mat daalo.
- Logout pe session destroy karo (‘req.session.destroy‘).

### 39.3.6 9.3.6 Extra Tips

- **Failed Login Logs:**
  - Har galat attempt ko log karo taaki suspicious activity track ho sake.
- **Account Lock Notification:**
  - User ko email bhejo agar account lock ho jaye taaki woh alert rahe.
- **Password Hashing:**
  - Bcrypt use karo passwords ko hash karne ke liye, salt ke saath.

## 39.4 9.4 Real-World Example in Hinglish

Maan lo ek e-commerce site pe login GET request se ho raha tha (‘/login?username=adminpassword=pa
Attacker ne URL intercept kiya aur credentials chura liye. Uske baad brute force script chalaya aur 1000 attempts mein password crack kar liya kyunki koi lockout policy nahi thi.

**Secure Fix:**

- Login ko POST pe shift kiya (‘/login‘ with body).

- Lockout policy lagayi—5 galat attempts pe 15 minute lock.
- HTTPS enable kiya taaki data encrypt rahe.
- Ab attacker ke liye hack karna mushkil ho gaya.

==========

# Secure Coding Guide: Express.js Vulnerabilities

**Secure Coding Guide: Express.js Vulnerabilities** Your Name March 28, 2025

## 40 Section: Cross-Site Scripting (XSS) in Express.js

### 40.1 1.1 XSS Kya Hai?

Bhai, Cross-Site Scripting (XSS) tab hota hai jab attacker koi malicious JavaScript code inject kar deta hai jo user ke browser mein chal jata hai. Yeh attack website ke input fields, URL parameters, ya database se aaye data ke through ho sakta hai. Agar XSS ho gaya, toh attacker user ka session chura sakta hai, fake pages dikha sakta hai, ya sensitive data (jaise cookies) le sakta hai.

**Types:**

- **Reflected XSS**: Jab malicious code URL mein hota hai aur server usko reflect karta hai.
- **Stored XSS**: Jab code database mein save ho jata hai aur har user ko dikhayi deta hai.
- **DOM-based XSS**: Jab client-side JavaScript input ko galat tarike se handle karta hai.

Ab main tujhe ek vulnerable aur secure code example deta hoon Express.js mein.

### 40.2 1.2 Vulnerable Code

**Listing: Vulnerable XSS Endpoint**

Listing 55: Vulnerable XSS Endpoint

```
const express = require('express');
const app = express();

app.get('/search', (req, res) => {
    const query = req.query.q; // User ka input URL se liya
    // Direct HTML mein input daal diya bina check ke
    res.send(`<h1>Search Results for: ${query}</h1>`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 56: Vulnerable XSS with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.get('/search', (req, res) => { // '/search' endpoint pe GET request
    handle kar raha hai
    const query = req.query.q; // URL se query parameter liya (?q=xyz)
    res.send(`<h1>Search Results for: ${query}</h1>`); // User input ko
        direct HTML mein daal diya, koi sanitization nahi
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 40.3 1.3 Kyun Vulnerable Hai?

- Yeh code user input (‘req.query.q‘) ko direct HTML mein daal raha hai bina kisi sanitization ke.
- Agar attacker URL mein ‘?q=¡script¿alert('hacked')¡/script¿‘ daal de, toh yeh browser mein execute ho jayega.
- Stored XSS ke case mein, agar yeh input DB mein save hua aur baad mein dikhaya gaya, toh har user ko attack hoga.
- Attacker user ke cookies chura sakta hai ya fake login page dikha sakta hai.

## 40.4 1.4 Secure Code

**Listing: Secure XSS Prevention**

Listing 57: Secure XSS Prevention

```
const express = require('express');
const sanitizeHtml = require('sanitize-html'); // HTML sanitization ke
   liye package
const app = express();

```

```
app.get('/search', (req, res) => {
    const query = req.query.q; // User ka input URL se liya
    // Input ko sanitize karke sirf safe HTML allow karo
    const safeQuery = sanitizeHtml(query, {
        allowedTags: [], // Koi tags allow nahi, plain text hi rahega
        allowedAttributes: {} // Koi attributes bhi nahi
    });
    res.send(`<h1>Search Results for: ${safeQuery}</h1>`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 58: Secure XSS with Comments

```
const express = require('express'); // Express framework import kiya
const sanitizeHtml = require('sanitize-html'); // Sanitize-html package
    import kiya, yeh malicious code hata dega
const app = express(); // Express app banaya
app.get('/search', (req, res) => { // '/search' endpoint pe GET request
    handle kar raha hai
    const query = req.query.q; // URL se query parameter liya (?q=xyz)
    const safeQuery = sanitizeHtml(query, { // Input ko sanitize kiya
        allowedTags: [], // Koi HTML tags allow nahi, sirf plain text
        allowedAttributes: {} // Koi attributes bhi allow nahi
    });
    res.send(`<h1>Search Results for: ${safeQuery}</h1>`); // Sanitized
        input ko HTML mein daala, ab safe hai
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 40.5 1.5 Kyun Secure Hai?

- ‘sanitize-html‘ package se input ko sanitize kiya gaya, toh ‘¡script¿‘ jaise tags remove ho jate hain.
- Ab attacker chahe ‘?q=¡script¿alert('hacked')¡/script¿‘ daale, yeh sirf text ke roop mein dikhega, execute nahi hoga.
- Yeh reflected aur stored XSS dono ko rokta hai jab tak output sanitize hota rahe.

## 40.6 1.6 Prevention Tips

- **Input Validation**: User input ko check karo ki woh expected format mein hai (e.g., regex se).
- **Output Escaping**: HTML mein daalne se pehle special characters escape karo (‘¡‘ → ‘¡‘).

- **CSP**: Content Security Policy lagao (‘res.set(’Content-Security-Policy’, ”script-src ’self’”)‘) taaki external scripts na chalein.
- **Tools**: ESLint plugin (‘eslint-plugin-no-unsanitized‘) use karo XSS dhoondhne ke liye.

# 41 Section: Cross-Site Request Forgery (CSRF) in Express.js

## 41.1 2.1 CSRF Kya Hai?

CSRF tab hota hai jab attacker user ko fake request karne pe majboor karta hai jab woh already logged-in hota hai. Jaise, agar tu bank app mein logged-in hai aur kisi malicious link pe click karta hai, toh attacker tere account se paisa transfer karwa sakta hai bina tere pata chale.

## 41.2 2.2 Vulnerable Code

**Listing: Vulnerable CSRF Endpoint**

Listing 59: Vulnerable CSRF Endpoint

```
const express = require('express');
const app = express();
app.use(express.urlencoded({ extended: true })); // Form data parse
    karne ke liye

app.post('/transfer', (req, res) => {
    const { amount, toAccount } = req.body; // Form se data liya
    // Koi CSRF check nahi, direct action perform kar raha hai
    res.send(`Transferred ${amount} to ${toAccount}, bhai!`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 60: Vulnerable CSRF with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.use(express.urlencoded({ extended: true })); // Form data ko parse
    karne ke liye middleware
app.post('/transfer', (req, res) => { // '/transfer' endpoint pe POST
    request handle kar raha hai
    const { amount, toAccount } = req.body; // Form se amount aur
        toAccount liya
    res.send(`Transferred ${amount} to ${toAccount}, bhai!`); // Koi
        CSRF check nahi, direct response bhej diya
```

```
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
    3000 port pe start kiya
```

## 41.3 2.3 Kyun Vulnerable Hai?

- Yeh code koi CSRF token check nahi karta, toh attacker ek fake form bana ke user ko submit karwa sakta hai.
- Agar user logged-in hai, toh browser cookies ke saath request bhej dega, aur transfer ho jayega.
- Example: Attacker ‘¡form action=”http://example.com/transfer” method=”POST”¿‘ bana ke ‘amount=1000‘ aur ‘toAccount=attacker‘ set kar sakta hai.

## 41.4 2.4 Secure Code

**Listing: Secure CSRF Protection**

Listing 61: Secure CSRF Protection

```
const express = require('express');
const csurf = require('csurf'); // CSRF protection ke liye package
const cookieParser = require('cookie-parser');
const app = express();

app.use(cookieParser()); // Cookies parse karne ke liye
app.use(express.urlencoded({ extended: true })); // Form data parse
    karne ke liye
app.use(csurf({ cookie: true })); // CSRF protection middleware

app.post('/transfer', (req, res) => {
    const { amount, toAccount } = req.body; // Form se data liya
    // CSRF token automatically check ho gaya middleware se
    res.send(`Transferred ${amount} to ${toAccount}, bhai!`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 62: Secure CSRF with Comments

```
const express = require('express'); // Express framework import kiya
const csurf = require('csurf'); // CSRF protection package import kiya
const cookieParser = require('cookie-parser'); // Cookies parse karne
    ke liye package
const app = express(); // Express app banaya
app.use(cookieParser()); // Cookies ko parse karne ke liye middleware
app.use(express.urlencoded({ extended: true })); // Form data parse
    karne ke liye middleware
```

```
app.use(csurf({ cookie: true })); // CSRF protection lagaya, token cookie mein store hoga
app.post('/transfer', (req, res) => { // '/transfer' endpoint pe POST request handle kar raha hai
    const { amount, toAccount } = req.body; // Form se amount aur toAccount liya
    res.send(`Transferred ${amount} to ${toAccount}, bhai!`); // CSRF token middleware ne check kar liya
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

### 41.5 2.5 Kyun Secure Hai?

- 'csurf' middleware har POST request pe CSRF token check karta hai, jo form mein hidden field ke roop mein hona chahiye.
- Agar token match nahi karta, toh request fail ho jayega (403 error).
- Attacker ke fake form mein token nahi hoga, toh attack kaam nahi karega.

### 41.6 2.6 Prevention Tips

- **CSRF Token**: Har form mein unique token daal ke server pe verify karo.
- **SameSite Cookies**: 'cookie: sameSite: 'strict' ' lagao taaki cross-site requests na chalein.
- **HTTP Method Check**: Sensitive actions ko POST/DELETE pe rakho, GET pe mat.

## 42 Section: Server-Side Request Forgery (SSRF) in Express.js

### 42.1 3.1 SSRF Kya Hai?

SSRF tab hota hai jab attacker server ko aisa URL fetch karne pe majboor karta hai jo internal systems ya sensitive resources tak jata hai, jaise 'http://localhost' ya 'http://internal-api'.

### 42.2 3.2 Vulnerable Code

**Listing: Vulnerable SSRF Endpoint**

Listing 63: Vulnerable SSRF Endpoint

```
const express = require('express');
const axios = require('axios'); // HTTP requests ke liye
const app = express();

app.get('/fetch', (req, res) => {
    const url = req.query.url; // User ka input URL se liya
    // Direct URL fetch kar rahe hain bina check ke
    axios.get(url).then(response => {
        res.send(response.data);
    }).catch(err => res.status(500).send('Kuch galat ho gaya'));
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 64: Vulnerable SSRF with Comments

```
const express = require('express'); // Express framework import kiya
const axios = require('axios'); // HTTP requests ke liye package
const app = express(); // Express app banaya
app.get('/fetch', (req, res) => { // '/fetch' endpoint pe GET request
   handle kar raha hai
    const url = req.query.url; // URL parameter se user input liya (?
       url=xyz)
    axios.get(url).then(response => { // Direct user ka URL fetch kiya,
        koi validation nahi
        res.send(response.data); // Response client ko bhej diya
    }).catch(err => res.status(500).send('Kuch galat ho gaya')); //
       Error handle kiya
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 42.3 3.3 Kyun Vulnerable Hai?

- Yeh code user ke diye URL ko direct fetch karta hai bina kisi check ke.
- Attacker '?url=http://localhost/admin' ya '?url=file:///etc/passwd' daal sakta hai aur sensitive data le sakta hai.
- Internal network ke resources (jaise DB ya admin panel) expose ho sakte hain.

## 42.4 3.4 Secure Code

**Listing: Secure SSRF Prevention**

Listing 65: Secure SSRF Prevention

```
const express = require('express');
const axios = require('axios');
```

```
const url = require('url'); // URL parsing ke liye
const app = express();

app.get('/fetch', (req, res) => {
    const inputUrl = req.query.url; // User ka input URL se liya
    const parsedUrl = url.parse(inputUrl); // URL ko parse kiya
    // Whitelist check: Sirf allowed domains
    const allowedDomains = ['api.example.com', 'www.example.com'];
    if (!allowedDomains.includes(parsedUrl.hostname)) {
        return res.status(403).send('Bhai, yeh URL allowed nahi hai!');
    }
    axios.get(inputUrl).then(response => {
        res.send(response.data);
    }).catch(err => res.status(500).send('Kuch galat ho gaya'));
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 66: Secure SSRF with Comments

```
const express = require('express'); // Express framework import kiya
const axios = require('axios'); // HTTP requests ke liye package
const url = require('url'); // URL ko parse karne ke liye Node.js
    module
const app = express(); // Express app banaya
app.get('/fetch', (req, res) => { // '/fetch' endpoint pe GET request
    handle kar raha hai
    const inputUrl = req.query.url; // URL parameter se user input liya
    const parsedUrl = url.parse(inputUrl); // URL ko parse kiya taaki
        hostname check kar sakein
    const allowedDomains = ['api.example.com', 'www.example.com']; //
        Allowed domains ki list
    if (!allowedDomains.includes(parsedUrl.hostname)) { // Check kiya
        ki URL ka hostname allowed hai ya nahi
        return res.status(403).send('Bhai, yeh URL allowed nahi hai!');
            // Agar nahi hai toh error
    }
    axios.get(inputUrl).then(response => { // Allowed URL ko fetch kiya
        res.send(response.data); // Response client ko bhej diya
    }).catch(err => res.status(500).send('Kuch galat ho gaya')); //
        Error handle kiya
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
    3000 port pe start kiya
```

## 42.5 3.5 Kyun Secure Hai?

- URL ko parse karke hostname check kiya gaya aur sirf whitelist domains allow kiye gaye.
- Attacker ab 'localhost' ya internal IPs fetch nahi kar sakta.
- Yeh SSRF attacks ko rokta hai kyunki invalid URLs reject ho jate hain.

## 42.6 3.6 Prevention Tips

- **Whitelist**: Sirf trusted domains allow karo.
- **Block Internal IPs**: ‘127.0.0.1‘, ‘10.0.0.0/8‘ jaise addresses block karo.
- **URL Validation**: ‘url.parse()‘ ya regex se URL check karo.

# 43 Section: File Inclusion/Path Traversal in Express.js

## 43.1 4.1 File Inclusion/Path Traversal Kya Hai?

Yeh tab hota hai jab attacker user input ke through server ke file system mein ghusta hai aur sensitive files (jaise ‘/etc/passwd‘) padh leta hai. Path traversal mein ‘../‘ use karke directory structure manipulate kiya jata hai.

## 43.2 4.2 Vulnerable Code

**Listing: Vulnerable Path Traversal**

Listing 67: Vulnerable Path Traversal

```
const express = require('express');
const fs = require('fs');
const app = express();

app.get('/file', (req, res) => {
    const fileName = req.query.name; // User ka input URL se liya
    // Direct file path bana diya bina check ke
    const filePath = `./files/${fileName}`;
    fs.readFile(filePath, 'utf8', (err, data) => {
        if (err) return res.status(500).send('Kuch galat ho gaya');
        res.send(data);
    });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 68: Vulnerable Path Traversal with Comments

```
const express = require('express'); // Express framework import kiya
const fs = require('fs'); // File system module import kiya
const app = express(); // Express app banaya
app.get('/file', (req, res) => { // '/file' endpoint pe GET request
   handle kar raha hai
    const fileName = req.query.name; // URL se file name liya (?name=
       xyz)
```

```
    const filePath = `./files/${fileName}`; // Direct file path bana diya, koi sanitization nahi
    fs.readFile(filePath, 'utf8', (err, data) => { // File ko read kiya
        if (err) return res.status(500).send('Kuch galat ho gaya'); // Error handle kiya
        res.send(data); // File ka data client ko bhej diya
    });
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

## 43.3 4.3 Kyun Vulnerable Hai?

- User input (`req.query.name`) ko direct file path mein use kiya gaya bina sanitization ke.
- Attacker `?name=../../etc/passwd` daal sakta hai aur sensitive files padh sakta hai.
- Koi base directory check nahi hai, toh file system ke bahar ja sakta hai.

## 43.4 4.4 Secure Code

**Listing: Secure Path Traversal Prevention**

Listing 69: Secure Path Traversal Prevention

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const app = express();

app.get('/file', (req, res) => {
    const fileName = req.query.name; // User ka input URL se liya
    // Base directory set kiya aur path normalize kiya
    const baseDir = path.resolve('./files');
    const filePath = path.join(baseDir, fileName);
    // Check kiya ki file base directory ke andar hai
    if (!filePath.startsWith(baseDir)) {
        return res.status(403).send('Bhai, bahar mat jao!');
    }
    fs.readFile(filePath, 'utf8', (err, data) => {
        if (err) return res.status(500).send('Kuch galat ho gaya');
        res.send(data);
    });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 70: Secure Path Traversal with Comments

```
const express = require('express'); // Express framework import kiya
const fs = require('fs'); // File system module import kiya
const path = require('path'); // Path manipulation ke liye module
const app = express(); // Express app banaya
app.get('/file', (req, res) => { // '/file' endpoint pe GET request handle kar raha hai
    const fileName = req.query.name; // URL se file name liya (?name=xyz)
    const baseDir = path.resolve('./files'); // Base directory ko absolute path mein set kiya
    const filePath = path.join(baseDir, fileName); // Safe path banaya, ../ ko handle karta hai
    if (!filePath.startsWith(baseDir)) { // Check kiya ki file base directory ke andar hai ya nahi
        return res.status(403).send('Bhai, bahar mat jao!'); // Agar bahar jata hai toh error
    }
    fs.readFile(filePath, 'utf8', (err, data) => { // File ko read kiya
        if (err) return res.status(500).send('Kuch galat ho gaya'); // Error handle kiya
        res.send(data); // File ka data client ko bhej diya
    });
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

### 43.5 4.5 Kyun Secure Hai?

- ‘path.resolve()‘ aur ‘path.join()‘ se path normalize hota hai, toh ‘../‘ kaam nahi karta.
- ‘startsWith‘ check se ensure kiya gaya ki file base directory ke andar hi rahe.
- Attacker ab system files access nahi kar sakta.

### 43.6 4.6 Prevention Tips

- **Sanitization**: ‘path.normalize()‘ use karo ‘../‘ hatane ke liye.
- **Base Directory**: Hamesha fixed base directory set karo.
- **Validation**: File names ko regex se check karo (e.g., ‘/[$a-zA-Z0-9-.$]+/‘).

## 44 Section: Logging and Monitoring Failures in Express.js

## 44.1 5.1 Logging and Monitoring Failures Kya Hai?

Yeh tab hota hai jab system mein proper logs ya monitoring nahi hoti, toh attacks detect nahi hote. Ethical hackers isko check karte hain ki failed logins, errors, ya suspicious activity track ho rahi hai ya nahi.

## 44.2 5.2 Vulnerable Code

**Listing: Vulnerable No Logging**

Listing 71: Vulnerable No Logging

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    // Koi logging nahi, direct check
    if (username === 'admin' && password === 'pass123') {
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        res.status(401).send('Galat credentials');
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 72: Vulnerable No Logging with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.post('/login', (req, res) => { // '/login' endpoint pe POST request
    handle kar raha hai
    const { username, password } = req.body; // Username aur password
       liya
    if (username === 'admin' && password === 'pass123') { // Direct
       check, koi log nahi
        res.json({ message: 'Login ho gaya, bhai!' }); // Success
           message
    } else {
        res.status(401).send('Galat credentials'); // Error message,
           koi log nahi
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 44.3 5.3 Kyun Vulnerable Hai?

- Failed login attempts ka koi log nahi hai, toh brute force attack pata nahi chalega.
- Success ya failure dono ka record nahi rakha ja raha.
- Agar attacker 1000 baar try kare, toh koi monitoring nahi hai usko pakadne ke liye.

## 44.4 5.4 Secure Code

**Listing: Secure Logging with Winston**

Listing 73: Secure Logging with Winston

```
const express = require('express');
const winston = require('winston'); // Logging ke liye package
const app = express();
app.use(express.json());

// Logger setup
const logger = winston.createLogger({
    level: 'info',
    format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.json()
    ),
    transports: [new winston.transports.File({ filename: 'app.log' })]
});

app.post('/login', (req, res) => {
    const { username, password } = req.body;
    const ip = req.ip; // IP address liya
    if (username === 'admin' && password === 'pass123') {
        logger.info(`Login successful for ${username} from ${ip}`); //
           Success log
        res.json({ message: 'Login ho gaya, bhai!' });
    } else {
        logger.warn(`Failed login attempt for ${username} from ${ip}`);
            // Failure log
        res.status(401).send('Galat credentials');
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 74: Secure Logging with Comments

```
const express = require('express'); // Express framework import kiya
const winston = require('winston'); // Winston logging package import
   kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
const logger = winston.createLogger({ // Logger configure kiya
```

```
    level: 'info', // Info level se logging hogi
    format: winston.format.combine( // Log format set kiya
        winston.format.timestamp(), // Timestamp add kiya
        winston.format.json() // JSON format mein logs
    ),
    transports: [new winston.transports.File({ filename: 'app.log' })]
       // Logs file mein save honge
});
app.post('/login', (req, res) => { // '/login' endpoint pe POST request
    handle kar raha hai
    const { username, password } = req.body; // Username aur password
       liya
    const ip = req.ip; // User ka IP address liya
    if (username === 'admin' && password === 'pass123') { //
       Credentials check
        logger.info(`Login successful for ${username} from ${ip}`); //
           Success ka log banaya
        res.json({ message: 'Login ho gaya, bhai!' }); // Success
           message
    } else {
        logger.warn(`Failed login attempt for ${username} from ${ip}`);
            // Failure ka log banaya
        res.status(401).send('Galat credentials'); // Error message
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 44.5 5.5 Kyun Secure Hai?

- `winston` se har login attempt (success ya failure) ka log banaya ja raha hai.
- IP address bhi track ho raha hai, toh suspicious activity monitor kar sakte ho.
- Logs `app.log` file mein save hote hain, jo analysis ke liye use ho sakte hain.

## 44.6 5.6 Prevention Tips

- **Log Everything**: Failed logins, errors, aur sensitive actions log karo.
- **No Sensitive Data**: Passwords ya tokens logs mein mat daalo.
- **Monitoring**: Logs ko ELK stack ya Splunk se monitor karo.

# 45 Section: Business Logic Flaws in Express.js

## 45.1 6.1 Business Logic Flaws Kya Hai?

Yeh tab hota hai jab app ka logic attacker ke favor mein manipulate ho jata hai, jaise price change karna ya unauthorized access lena. Yeh code bug nahi, balki design ya logic ki kami hoti hai.

## 45.2 6.2 Vulnerable Code

**Listing: Vulnerable Price Manipulation**

Listing 75: Vulnerable Price Manipulation

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/buy', (req, res) => {
    const { itemId, price } = req.body; // Client se price liya
    // Direct client ka price use kiya bina check ke
    res.send(`Bought item ${itemId} for ${price}, bhai!`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 76: Vulnerable Price Manipulation with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.post('/buy', (req, res) => { // '/buy' endpoint pe POST request
   handle kar raha hai
    const { itemId, price } = req.body; // Client se itemId aur price
       liya
    res.send(`Bought item ${itemId} for ${price}, bhai!`); // Client ka
        price direct use kiya, koi server check nahi
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
   3000 port pe start kiya
```

## 45.3 6.3 Kyun Vulnerable Hai?

- Client se aaya `price` direct use ho raha hai, toh attacker `price=0` ya negative value bhej sakta hai.
- Server pe koi validation nahi hai ki price sahi hai ya nahi.
- Yeh business logic flaw hai kyunki app trust kar raha hai client pe.

## 45.4 6.4 Secure Code

**Listing: Secure Price Validation**

Listing 77: Secure Price Validation

```
const express = require('express');
const app = express();
app.use(express.json());

// Dummy DB for item prices
const items = { 1: 100, 2: 200 };

app.post('/buy', (req, res) => {
    const { itemId, price } = req.body; // Client se data liya
    const actualPrice = items[itemId]; // Server se real price liya
    if (!actualPrice || price !== actualPrice) { // Price match check kiya
        return res.status(403).send('Bhai, price galat hai!');
    }
    res.send(`Bought item ${itemId} for ${actualPrice}, bhai!`);
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 78: Secure Price Validation with Comments

```
const express = require('express'); // Express framework import kiya
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
const items = { 1: 100, 2: 200 }; // Dummy DB jahan real prices store hain
app.post('/buy', (req, res) => { // '/buy' endpoint pe POST request handle kar raha hai
    const { itemId, price } = req.body; // Client se itemId aur price liya
    const actualPrice = items[itemId]; // Server se item ka asli price liya
    if (!actualPrice || price !== actualPrice) { // Check kiya ki price match karta hai ya nahi
        return res.status(403).send('Bhai, price galat hai!'); // Agar galat hai toh error
    }
    res.send(`Bought item ${itemId} for ${actualPrice}, bhai!`); // Real price ke saath response
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

## 45.5 6.5 Kyun Secure Hai?

- Price ko server-side DB se check kiya gaya, client ka input ignore kiya.

- Attacker ab price manipulate nahi kar sakta kyunki server pe validation hai.
- Yeh logic flaw ko fix karta hai.

### 45.6 6.6 Prevention Tips

- **Server-Side Validation**: Sensitive values (price, quantity) client pe trust mat karo.
- **DB Checks**: Har action ke liye DB ya config se data verify karo.
- **Rate Limiting**: Unlimited purchases rokne ke liye limit lagao.

## 46 Section: API Security in Express.js

### 46.1 7.1 API Security Kya Hai?

API security mein REST ya GraphQL endpoints ko secure karna hota hai taaki unauthorized access, data leaks, ya abuse na ho. Common issues hain broken auth, no rate limiting, aur input validation ki kami.

### 46.2 7.2 Vulnerable Code

**Listing: Vulnerable API Endpoint**

Listing 79: Vulnerable API Endpoint

```
const express = require('express');
const app = express();
app.use(express.json());

app.get('/api/data', (req, res) => {
    const apiKey = req.query.key; // API key URL se liya
    // Weak API key check
    if (apiKey === '12345') {
        res.json({ data: 'Secret data, bhai!' });
    } else {
        res.status(401).send('Galat key');
    }
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 80: Vulnerable API with Comments

```
const express = require('express'); // Express framework import kiya
```

```
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
app.get('/api/data', (req, res) => { // '/api/data' endpoint pe GET request handle kar raha hai
    const apiKey = req.query.key; // API key URL se liya (?key=xyz)
    if (apiKey === '12345') { // Weak aur predictable key check
        res.json({ data: 'Secret data, bhai!' }); // Data bhej diya
    } else {
        res.status(401).send('Galat key'); // Error message
    }
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

## 46.3 7.3 Kyun Vulnerable Hai?

- API key weak aur predictable hai (‘12345‘), guess karna aasan hai.
- Key URL mein expose ho raha hai, logs mein leak ho sakta hai.
- Koi rate limiting nahi hai, toh brute force ho sakta hai.

## 46.4 7.4 Secure Code

**Listing: Secure API Endpoint**

Listing 81: Secure API Endpoint

```
const express = require('express');
const rateLimit = require('express-rate-limit');
const app = express();
app.use(express.json());

// Rate limiter
const apiLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minute window
    max: 100 // 100 requests per IP
});

// Dummy valid API key
const validApiKey = 'x7k9p2m4q8r5t1n3';

app.get('/api/data', apiLimiter, (req, res) => {
    const apiKey = req.headers['x-api-key']; // Header se key liya
    if (apiKey !== validApiKey) { // Strong key check
        return res.status(401).send('Galat key, bhai!');
    }
    res.json({ data: 'Secret data, bhai!' });
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 82: Secure API with Comments

```
const express = require('express'); // Express framework import kiya
const rateLimit = require('express-rate-limit'); // Rate limiting package
const app = express(); // Express app banaya
app.use(express.json()); // JSON body parse karne ke liye
const apiLimiter = rateLimit({ // Rate limiter configure kiya
    windowMs: 15 * 60 * 1000, // 15 minute ka window
    max: 100 // 100 requests allow honge per IP
});
const validApiKey = 'x7k9p2m4q8r5t1n3'; // Strong random API key
app.get('/api/data', apiLimiter, (req, res) => { // '/api/data' endpoint pe GET request
    const apiKey = req.headers['x-api-key']; // API key header se liya, URL mein nahi
    if (apiKey !== validApiKey) { // Strong key ke saath check
        return res.status(401).send('Galat key, bhai!'); // Error message
    }
    res.json({ data: 'Secret data, bhai!' }); // Data bhej diya
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

## 46.5 7.5 Kyun Secure Hai?

- API key header mein liya gaya (‘x-api-key‘), toh URL mein expose nahi hota.
- Strong random key use kiya, guess karna mushkil hai.
- ‘rateLimit‘ se brute force attacks rukenge.

## 46.6 7.6 Prevention Tips

- **JWT**: Complex APIs ke liye JWT tokens use karo.
- **Rate Limiting**: Har endpoint pe limit lagao.
- **Input Validation**: API inputs ko strictly check karo.

# 47 Section: Dependency Injection/Supply Chain Attacks in Express.js

## 47.1 8.1 Dependency Injection/Supply Chain Attacks Kya Hai?

Yeh tab hota hai jab koi malicious npm package ya dependency install ho jati hai jo code mein backdoor ya data leak kar sakti hai.

## 47.2 8.2 Vulnerable Code

**Listing: Vulnerable Dependency Usage**

Listing 83: Vulnerable Dependency Usage

```
const express = require('express');
const badPackage = require('malicious-package'); // Fake malicious
    package
const app = express();

app.get('/', (req, res) => {
    badPackage.run(); // Malicious code chal sakta hai
    res.send('Hello, bhai!');
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 84: Vulnerable Dependency with Comments

```
const express = require('express'); // Express framework import kiya
const badPackage = require('malicious-package'); // Fake malicious
    package import kiya
const app = express(); // Express app banaya
app.get('/', (req, res) => { // '/' endpoint pe GET request handle kar
    raha hai
    badPackage.run(); // Malicious package ka function call kiya, yeh
        kuch bhi kar sakta hai
    res.send('Hello, bhai!'); // Response bhej diya
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server
    3000 port pe start kiya
```

## 47.3 8.3 Kyun Vulnerable Hai?

- Unknown package (‘malicious-package‘) use kiya gaya jo data chura sakta hai ya server crash kar sakta hai.
- Package verify nahi kiya gaya install se pehle.
- Supply chain attack ka risk hai.

## 47.4 8.4 Secure Code

**Listing: Secure Dependency Management**

Listing 85: Secure Dependency Management

```
const express = require('express');
const app = express();

// package.json mein lock kiya hua trusted version
// "express": "^4.18.2" aur package-lock.json use kiya

app.get('/', (req, res) => {
    res.send('Hello, bhai!'); // Sirf trusted code
});

app.listen(3000, () => console.log('Server chal raha hai'));
```

**Line-by-Line Explanation with Comments:**

Listing 86: Secure Dependency with Comments

```
const express = require('express'); // Express framework import kiya, trusted version
const app = express(); // Express app banaya
app.get('/', (req, res) => { // '/' endpoint pe GET request handle kar raha hai
    res.send('Hello, bhai!'); // Sirf trusted code chal raha hai, koi extra package nahi
});
app.listen(3000, () => console.log('Server chal raha hai')); // Server 3000 port pe start kiya
```

## 47.5 8.5 Kyun Secure Hai?

- Sirf trusted aur verified package (‘express‘) use kiya gaya.
- ‘package-lock.json‘ se exact versions lock kiye gaye, toh malicious update nahi aayega.
- Extra dependencies avoid ki gayi.

## 47.6 8.6 Prevention Tips

- **Audit**: ‘npm audit‘ chalao har dependency check karne ke liye.
- **Lock File**: ‘package-lock.json‘ commit karo repo mein.
- **Trusted Sources**: Sirf official npm packages use karo.

# 48 Section: Secure Coding Best Practices Summary

## 48.1 9.1 Secure Coding Best Practices Kya Hai?

Yeh ek quick summary hai jo har ethical hacker aur developer ko yaad rakhna chahiye. Yeh tere notes ka conclusion bhi ban sakta hai.

## 48.2 9.2 Best Practices

- **Input Validation**: Har user input ko check karo (regex, type checking).
- **Output Escaping**: HTML, JS mein data daalne se pehle escape karo.
- **Least Privilege**: Server ko minimum permissions do (e.g., read-only DB user).
- **Defense-in-Depth**: Multiple layers of security lagao (auth, rate limiting, encryption).
- **Tools**: 'eslint-plugin-security', OWASP ZAP, Burp Suite use karo testing ke liye.
- **Checklist**: Production se pehle default accounts hatao, HTTPS enable karo, logs setup karo.

## 48.3 9.3 Hinglish Summary

"Bhai, hamesha input check karo, output escape karo, server ko kam power do, aur tools jaise Burp Suite se test karo. Production mein jaane se pehle sab double-check kar lena!"

# 49 Conclusion

This guide covered **XSS, CSRF, SSRF, Path Traversal, Logging Failures, Business Logic Flaws, API Security, aur Dependency Attacks** in Express.js with vulnerable and secure examples. **Key takeaways** include input validation, output sanitization, proper logging, and dependency management. Ab tu in vulnerabilities ko spot aur fix kar sakta hai, bhai!

# Vulnerable Machines for Practice and Source Code Review Step by Step process followed by Hackers...

**Vulnerable Machines for Practice and Source Code Review Step by Step process followed by Hackers...** March 28, 2025

## 50 Vulnerable machines for practice

### 50.1 2. WebGoat (OWASP Project)

- **Kya Hai?** Yeh OWASP ka ek vulnerable web app hai jo Java pe based hai aur real-world vulnerabilities sikhaane ke liye banaya gaya hai.
- **GitHub Link:** https://github.com/WebGoat/WebGoat
- **Kaise Use Kare?**
  - Isko Docker ya JAR file se run kar sakta hai (Java install hona chahiye).
  - Pehle black-box se attack kar—har lesson mein ek challenge hota hai (jaise SQL Injection ya Broken Authentication).
  - Hint ya solution chahiye toh source code (Java files) khol ke dekh—har vulnerability ka explanation milta hai.
- **Fayda:** Yeh thoda advanced hai aur source code review ke liye bhi detailed hai. Real-world scenarios cover karta hai.

### 50.2 1. Juice Shop (OWASP Project)

- **Kya Hai?** Yeh ek modern vulnerable web app hai jo Node.js pe based hai aur OWASP Top 10 vulnerabilities cover karta hai.
- **GitHub Link:** https://github.com/juice-shop/juice-shop
- **Kaise Use Kare?**
  - Isko Docker ya npm se run kar sakta hai.
  - Black-box se shuru kar—XSS, SQL Injection, ya Broken Access Control try kar.
  - Source code (JavaScript/Node.js files) khol ke padh—har vulnerability ka logic clearly samajh aata hai.
- **Fayda:** Yeh modern tech stack pe kaam karta hai aur code review ke liye bahut acha hai. CTF-style challenges bhi hain.

## 50.3 Practical Approach Kaise Lena Hai?

1. **Black-Box Pehle:**
   - Machine ko run kar aur pehle bina source code dekhe hack karne ki koshish kar.
   - Tools jaise Burp Suite, nmap, Metasploit, ya manual payloads use kar.
   - Goal rakho ki kam se kam ek vulnerability exploit kar sake.
2. **Agar Nahi Hua Toh Source Code:**
   - GitHub repo ya VM ke files khol aur code padh.
   - Vulnerable function calls (jaise 'eval()', 'system()', 'mysql$_{q}uery$()')$dhoondh.Samajhkebaaddobara$
3. **Notes Banao:**
   - Har machine ke baad likh ki kya seekha—black-box se kya mila aur source code se kya samajh aaya.
   - Yeh tere job mein code review ke liye bahut kaam aayega.

# 51 General Source Code Review Technique Step-by-Step (Hacker's Approach)

Alright, bhai! Tujhe source code review ka detailed step-by-step tarika chahiye jo ek hacker follow karta hai, especially jab koi company apna project deke bolti hai, "Bhai, iska code review kar do." Main tujhe ek general approach dunga jo real-world mein kaam aayega, aur isme login functionality ka example bhi mix kar dunga taaki tujhe samajh aaye. Yeh approach black-box aur white-box dono ko cover karega, kyunki tu shuruat mein black-box try karega aur agar nahi hua toh source code padhega. Chalo shuru karte hain!

## 51.1 Step 1: Project Ko Samajhna (Reconnaissance)

- **Kya Karna Hai?** Pehle project ka basic idea samajh—yeh kya hai? Web app hai, API hai, ya koi aur cheez? Company se basic info le:
  - Tech stack kya hai? (e.g., Express.js, PHP, Django)
  - Kya-kya features hain? (e.g., login, file upload, payments)
  - Koi specific area jahan focus chahiye? (e.g., login functionality)
- **Login Example:** Maan le company boli, "Hamara login page check karo." Toh pehle samajh ki login ka flow kaisa hai—username/password ya OAuth? Frontend mein form hai ya API call?

- **Tools:**
  - Documentation (agar diya ho) padh.
  - Browser mein app khol ke dekho (black-box style).
- **Goal:** High-level understanding banao taaki code padhne mein context clear ho.

## 51.2 Step 2: Black-Box Testing Se Shuruat

- **Kya Karna Hai?** Bina code dekhe pehle app ko hack karne ki koshish karo. Yeh real-world mein common hai kyunki hacker ko shuru mein code nahi milta.
  - Input fields dhoondho (e.g., email, password).
  - Common attacks try karo:
    - SQL Injection: '' OR 1=1 –'
    - XSS: '¡script¿alert(1)¡/script¿'
    - CSRF: Fake form bana ke POST request bhejo.
    - Brute Force: Password guess karo (e.g., 'admin:admin').
- **Login Example:** Login form pe email mein '' OR 1=1 –' daal ke dekho—kya login ho jata hai? Password field mein '¡script¿' try karo—kya error aata hai ya execute hota hai?
- **Tools:**
  - Burp Suite (requests intercept karo).
  - Postman (API endpoints test karo).
  - Browser DevTools (network calls dekho).
- **Goal:** Weak points dhoondho aur note karo ki kahan attack kaam karta hai ya fail hota hai. Agar fail hua, toh yeh hint hai ki code mein kya check karna hai.

## 51.3 Step 3: Source Code Access Ke Baad White-Box Shuru

- **Kya Karna Hai?** Jab black-box se kuch nahi mila ya company ne code de diya, toh ab source code khol ke padhna shuru karo.
  - Codebase ka structure samajho:
    - Main files kahan hain? (e.g., 'app.js', 'routes/', 'controllers/')
    - Configuration files kahan hain? (e.g., '.env', 'config.js')
  - Vulnerable areas pe focus karo jo black-box mein try kiye (e.g., login).
- **Login Example:** Login ka endpoint dhoondho (e.g., 'app.post('/login', ...)'). Yeh file mein check karo ki email aur password ka input kaise liya gaya hai.
- **Tools:**

  - VS Code (code search ke liye).
  - 'grep' ya 'find' commands (specific keywords dhoondhne ke liye).

- **Goal:** Code ka flow samajhna aur entry points (user input) dhoondhna.

## 51.4 Step 4: Input Handling Aur Validation Check Karna

- **Kya Karna Hai?** Har jagah jahan user input aata hai (query params, body, headers), usko trace karo aur dekho:
  - Input kaise liya gaya hai? (e.g., 'req.body.email', 'req.query.id')
  - Kya sanitization ya validation hai? (e.g., regex, length check)
  - Kya direct database ya system call mein ja raha hai?
- **Login Example:** Code mein dekho:

Listing 87: Vulnerable Login Input Handling

```
app.post('/login', (req, res) => {
    const { email, password } = req.body; // Input yahan se liya
    const query = `SELECT * FROM users WHERE email = '${email}' AND
        password = '${password}'`; // Direct concatenation?
});
```

  - Yeh vulnerable hai kyunki email/password direct query mein ja raha hai—SQL Injection possible hai.
  - Secure version mein parameterized query hona chahiye:

Listing 88: Secure Login Input Handling

```
db.query('SELECT * FROM users WHERE email = ? AND password = ?'
    , [email, password]);
```

- **Dhoondhne Wali Cheezein:**
  - No validation (e.g., email ka regex nahi).

– Unsafe string concatenation (e.g., '*email*').*Missingescaping/encoding.*

- **Goal:** Input handling ke flaws dhoondho jo injection ya bypass allow kar sakein.

## 51.5 Step 5: Authentication Aur Authorization Check

- **Kya Karna Hai?** Dekho ki app user ko kaise authenticate aur authorize karta hai:
  - Login ke baad session ya token kaise manage hota hai? (e.g., JWT, cookies)
  - Sensitive endpoints pe permission check hai ya nahi? (e.g., '/admin')

  - Weak credentials allowed hain? (e.g., 'admin:admin')

- **Login Example:** Code mein dekho:

Listing 89: Vulnerable Authentication

```
if (email === 'admin' && password === 'pass123') {
    res.send('Logged in!');
}
```

  - Yeh weak hai kyunki hardcoded credentials hain aur koi hash nahi.
  - Secure version mein bcrypt hona chahiye:

Listing 90: Secure Authentication

```
const hash = await bcrypt.hash(password, 10);
if (await bcrypt.compare(password, storedHash)) { ... }
```

  - Authorization check bhi dhoondho:

```
if (req.user.role !== 'admin') { res.status(403).send('Access
    denied'); }
```

- **Dhoondhne Wali Cheezein:**
  - Hardcoded credentials.
  - Missing role checks.
  - Weak session management (e.g., predictable tokens).
- **Goal:** Authentication bypass ya privilege escalation ke risks dhoondho.

## 51.6 Step 6: Sensitive Data Exposure Check

- **Kya Karna Hai?** Dekho ki sensitive data (passwords, API keys, PII) kaise handle hota hai:
  - Kya encryption hai? (e.g., AES-256)
  - Kya logs mein leak ho raha hai? (e.g., 'console.log(password)')

– Kya hardcoded secrets hain? (e.g., 'const $\text{API}_KEY =' xyz'$')

- **Login Example:** Code mein dekho:

Listing 91: Vulnerable Data Exposure

```
console.log(`Login attempt: ${email}, ${password}`);
```

  - Yeh galat hai kyunki password logs mein leak ho sakta hai.
  - Secure mein sensitive data log nahi karna:

Listing 92: Secure Data Handling

```
logger.info(`Login attempt for ${email}`);
```

- **Dhoondhne Wali Cheezein:**
  - Weak ciphers (e.g., DES, MD5).
  - Hardcoded keys in code.
  - Unencrypted data transmission (HTTP ke bajaye HTTPS).
- **Goal:** Data leaks ya weak encryption dhoondho.

## 51.7 Step 7: Business Logic Aur Edge Cases

- **Kya Karna Hai?** App ke logic ko samajho aur dekho ki kahan attacker usko manipulate kar sakta hai:
  - Kya price, quantity ya status client se aata hai aur server check nahi karta?
  - Kya rate limiting hai ya unlimited requests allowed hain?
- **Login Example:** Agar login attempts limit nahi hai:

Listing 93: Vulnerable Logic

```
app.post('/login', (req, res) => { ... }); // No rate limit
```

  - Yeh brute force ke liye vulnerable hai.
  - Secure mein rate limiting add karo:

Listing 94: Secure Logic

```
const loginLimiter = rateLimit({ windowMs: 15 * 60 * 1000, max:
    5 });
app.post('/login', loginLimiter, (req, res) => { ... });
```

- **Dhoondhne Wali Cheezein:**
  - Client-controlled values (e.g., price=0).
  - Missing rate limits.
  - Logical flaws (e.g., negative values allowed).
- **Goal:** Logic flaws jo app ke purpose ko break kar sakein.

## 51.8 Step 8: Dependencies Aur Third-Party Code

- **Kya Karna Hai?** Check karo ki app mein kaunsi external libraries ya packages use hue hain:
  - 'package.json' ya 'requirements.txt' dekho.
  - Known vulnerabilities ke liye check karo (e.g., 'npm audit').
  - Third-party code mein suspicious logic dhoondho.

- **Login Example:** Agar koi outdated package hai:

```
"express": "4.0.0" // Old version with known bugs
```

  - Isko latest version pe update karna chahiye.

- **Tools:**
  - 'npm audit' (Node.js).
  - Snyk ya Dependabot (vulnerability scanning).
- **Goal:** Supply chain attacks ya outdated code ke risks dhoondho.

## 51.9 Step 9: Findings Document Karna

- **Kya Karna Hai?** Har vulnerability ko note karo aur report banao:
  - Issue kya hai? (e.g., SQL Injection in login)
  - Code ka snippet (vulnerable line).
  - Impact kya hoga? (e.g., unauthorized access)
  - Fix ka suggestion (e.g., parameterized query).
- **Login Example:**

```
Vulnerability: SQL Injection in /login endpoint
Code: `SELECT * FROM users WHERE email = '${email}'`
Impact: Attacker can bypass login with ' OR 1=1 --
Fix: Use `db.query('SELECT * FROM users WHERE email = ?', [email])`
```

- **Goal:** Company ko clear aur actionable report dena.

## 51.10 Step 10: Verification Aur Retesting

- **Kya Karna Hai?** Fixes ke baad dobara test karo:
  - Black-box se confirm karo ki attack ab kaam nahi karta.
  - Code mein check karo ki suggestion apply hua ya nahi.
- **Login Example:** Agar parameterized query add hua, toh '' OR 1=1 –' try karo—ab login nahi hona chahiye.
- **Goal:** Ensure karna ki vulnerabilities fix ho gayi hain.

### 51.11 Tera Practice Plan

1. **Notes Se Shuru Kar:** Tere notes (jo tune upload kiye) mein examples hain—jaise login ka vulnerable code ('SELECT * FROM users WHERE email = '$email$'). $Inkopadhaursamaj$ $boxkaro(e.g., loginbypass), phircodekholkedekhokikyunvulnerabletha.$

2. **Chhota Project Bana:** Ek simple Express.js app bana (login wala) aur jaan bujh ke flaws daal—phir upar wale steps follow karke review kar.

3. **Tools Use Kar:** Burp Suite, Postman, VS Code jaise tools ke saath practice kar.

### 51.12 General Tips

- **Pattern Dhoondho:** Har language mein common flaws hote hain (e.g., Express.js mein no rate limiting, direct input usage).
- **Hinglish Mein Socho:** "Bhai, yeh input kahan ja raha hai? Kya yeh query mein direct mix ho raha hai?"
- **Patience Rakho:** Shuruat mein time lagega, lekin practice se speed aayegi.

## 52 Conclusion

This guide covers **vulnerable machines** like WebGoat and Juice Shop for practice, plus a **hacker's step-by-step source code review approach**. Ab tu black-box se shuru karke white-box tak ja sakta hai aur real-world vulnerabilities dhoondh sakta hai, bhai!

==========