BEST PRACTICES TO SECURE WEB APPLICATIONS

Ronak Hingonia

IIT Gandhinagar

ronakhingonia1@gmail.com

I. INTRODUCTION

This document is intended for students and professional developers as a quick reference checklist for security practices. It covers 11 categorized security aspects that, when implemented, can help prevent the corresponding attacks. A before-and-after code example is then presented to demonstrate what secure code looks like in Express.js for easier understanding. Finally, a table lists the top open-source security tools every developer should know.

II. KEY TERMS

Authentication, Authorization, Encryption, Firewall, Web Application, Vulnerability, Attack

III. DEFINITIONS

1. Authentication: The process of verifying the identity of a user or system.
2. Authorization: The process of restricting user access to resources based on roles or permissions, ensuring that users can only perform actions within their privileges.
3. Encryption: The technique of converting information into a secure format.
4. Firewall: A network security system that monitors and controls incoming and outgoing traffic.
5. Web Application: A software application that runs on a web server and is accessed via a web browser.
6. Vulnerability: A weakness in a system that can be exploited to cause harm.
7. Attack: An information security threat that involves an attempt to obtain, alter, destroy, remove, implant, or reveal unauthorized information.

IV. ABBREVIATIONS

- SQL: Structured Query Language
- XSS: Cross-Site Scripting
- MFA: Multi-Factor Authentication
- 2FA: Two-Factor Authentication
- RBAC: Role-Based Access Control
- ABAC: Attribute-Based Access Control
- TLS: Transport Layer Security
- HSTS: HTTP Strict Transport Security
- DoS: Denial-of-Service
- DTO: Data Transfer Object
- CORS: Cross-Origin Resource Sharing
- OSS: Open Source Software
- CI/CD: Continuous Integration / Continuous Deployment
- IDE: Integrated Development Environment
- SBOM: Software Bill of Materials
- CLI: Command-Line Interface
- ZAP: Zed Attack Proxy
- OSV: Open Source Vulnerabilities
- KICS: Keeping Infrastructure as Code Secure
- OPA: Open Policy Agent
- IaC: Infrastructure as Code

S No.	CATEGORY	DESCRIPTION	IMPLEMENTATION/ PRACTICES	PREVENTS FROM
1	**Authentication**	Verifies the identity of users or systems to ensure that only legitimate actors gain access to the application.	• Enforce multi-factor authentication (MFA) or two-factor (2FA) • Use secure hashing algorithms for storing passwords • Integrate biometric or OTP solutions	• Unauthorized access • Identity spoofing • Brute force attacks
2	**Authorization**	Restricts user access based on roles or attributes, minimizing damage from compromised accounts. Critical for compliance and data integrity	• Deny by default • Never trust the request • Follow Least privilege • Use role-based (RBAC) or attribute based access control (ABAC) (preferred) • Decouple from logic: use Custom security expressions • No hard-coding • Secure APIs with token validation (authentication)	• Unauthorized resource access • Role Explosion • Broken object level Authorization • Data breaches • Misuse of system functionality
3	**Data Control**	Prevents Excessive Data Exposure—a vulnerability where more information than necessary is sent to the client, inadvertently exposing sensitive data. This type of problem is often framed as Broken Object Property Level Authorization (BOPLA)	• Create Data Transfer Objects (DTOs) • Least privilege • Select only the fields that are safe and necessary for the client. • A mapper library can be used instead of manually created DTOs.	• Property Level issues • Unauthorized disclosure of sensitive data
4	**Input Validation**	Ensures that user-provided data conforms to expected formats and rules before processing, preventing malicious input from being processed by the application.	• Never trust the request • Validate type, length, format, and range • Enforce Limits • Validate Strings (use regexp) • Prefer allowed lists (deny by deafault) • Validate Parameters • Should add the same validations on frontend as well	• SQL Injection • XSS (cross-site scripting) • Command injection. • Exceptions - might expose the tech stack being used in back end which can be then used by a malicious actor.
5	**File Upload**	File uploads can be a significant security risk if not handled correctly. Attackers can upload malicious files that can compromise the security of the entire application	• Scan the file first for viruses • Never trust the request • Use upload & download limits • Do not trust the content type headers because that can be altered as well. • Validate the extension and type of file • Set file name length limit • Always check for file MetaData directly	• Path traversal vulnerabilities • Malware Injection • Performance issues • Remote code execution

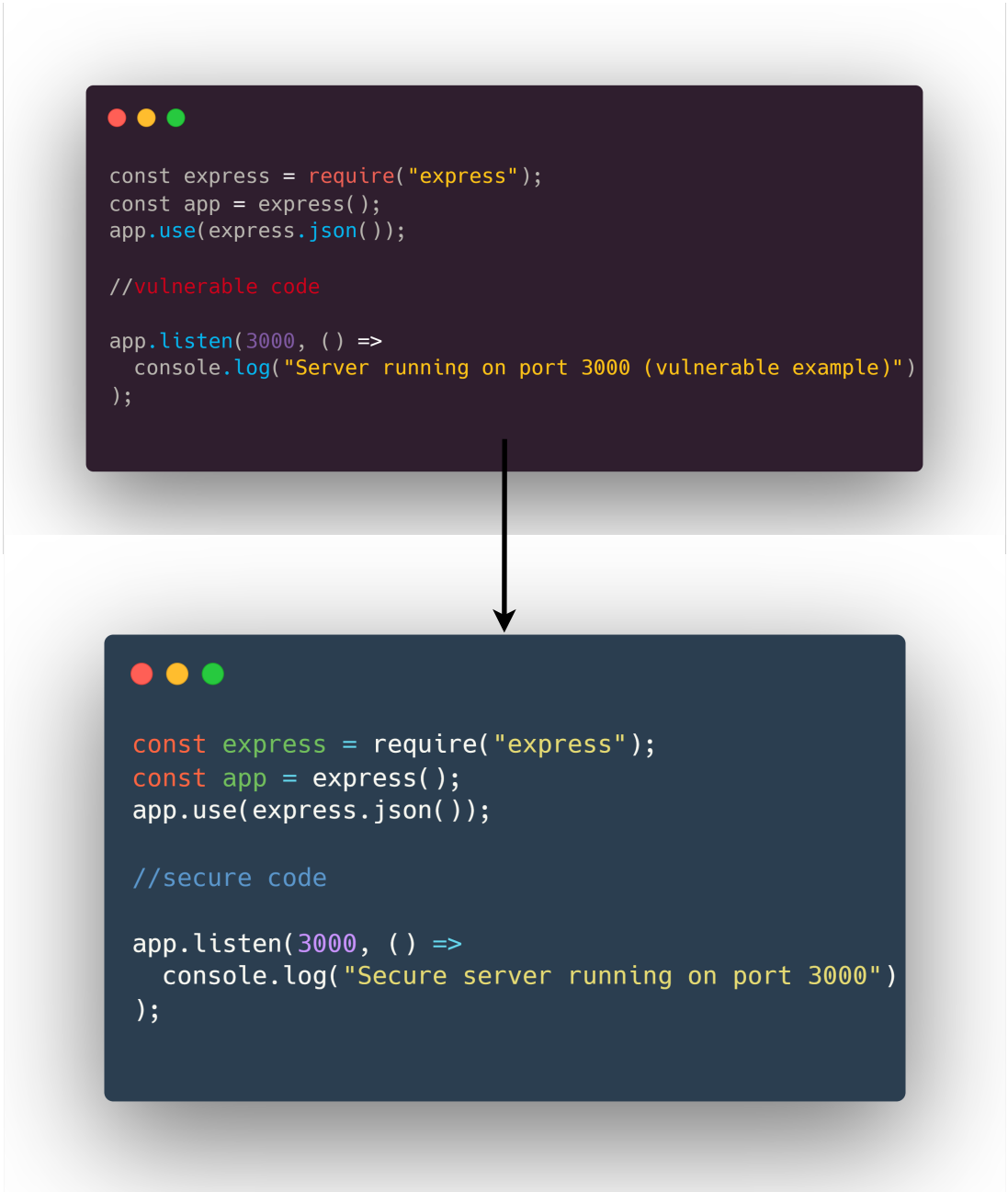
S No.	CATEGORY	DESCRIPTION	IMPLEMENTATION/ PRACTICES	PREVENTS FROM
6	**Cross Origin Resource Sharing**	Manages how web applications can request resources from a different domain, ensuring secure cross-origin communication.	• Configure CORS policies to allow only trusted domains • Use proper HTTP headers (e.g., `Access-Control-Allow-Origin`) • Restrict methods (e.g., GET, POST) and headers allowed in cross-origin requests • Avoid using wildcards (`*`) for sensitive endpoints • Implement preflight request handling for complex requests • Validate and sanitize incoming cross-origin requests	• Unauthorized cross-origin data access • Data leakage to untrusted domains
7	**Session Management**	Ensures secure handling of user sessions to prevent unauthorized access or session hijacking.	• Use strong & random session IDs - long and complex. • Implement session expiration and timeout policies • Regenerate session IDs after login or performs any sensitive actions. • Use HTTPS for session cookies	• Session hijacking • Session fixation • Unauthorized Data access
8	**Rate Limiting**	Controls the frequency of client requests within a defined time frame to protect application resources from abuse and overload	• Use algorithms like token bucket, leaky bucket, fixed window, or sliding window counters to control the flow of requests. • Use Distributed Rate Limiting to ensure consistent enforcement across multiple servers • Implement at multiple layers (API gateway, load balancer, or web server level)	• Denial-of-service (DoS) attacks • Brute-force attempts • API abuse • Resource exhaustion
9	**Secure Communication**	Ensures data transmitted between the client and server is encrypted and protected from interception.	• Use HTTPS with strong TLS configurations • Enforce HSTS (HTTP Strict Transport Security) • Disable weak ciphers and protocols	• Man-in-the-middle attacks • Data interception • Eavesdropping
10	**Web Application Firewalls (WAFs)**	Acts as a protective barrier between the web application and incoming traffic by filtering, monitoring, and blocking malicious requests based on pre-defined security rules.	• Deploy a WAF (hardware, software, or cloud-based) • Configure custom security rules and anomaly detection	• Automated attacks • Cross-site scripting • SQL injection • Denial of service
11	**Third-Party Dependencies**	Manages risks associated with third-party libraries and services used in the application.	• Regularly update third-party libraries • Monitor for vulnerabilities in dependencies • Use trusted sources for libraries • Limit permissions granted to third-party services	• Exploitation of library vulnerabilities • Supply chain attacks

Secure Coding Practices (JS)

This document outlines secure coding practices in JavaScript and Express by demonstrating common security vulnerabilities and their corresponding solutions. Each example is presented in a "before-and-after" format to highlight the transition from insecure to secure code.

- The first code represents an insecure version with vulnerabilities, while the last code is its secure, finalized version.
- Intermediate steps, if any, are shown with different themes are to illustrate the progression toward secure implementation.
- Please go through the comments in code for explanation.

This visual approach is intended to make it easier to understand the security issues and how to address them effectively.



1. Authorization

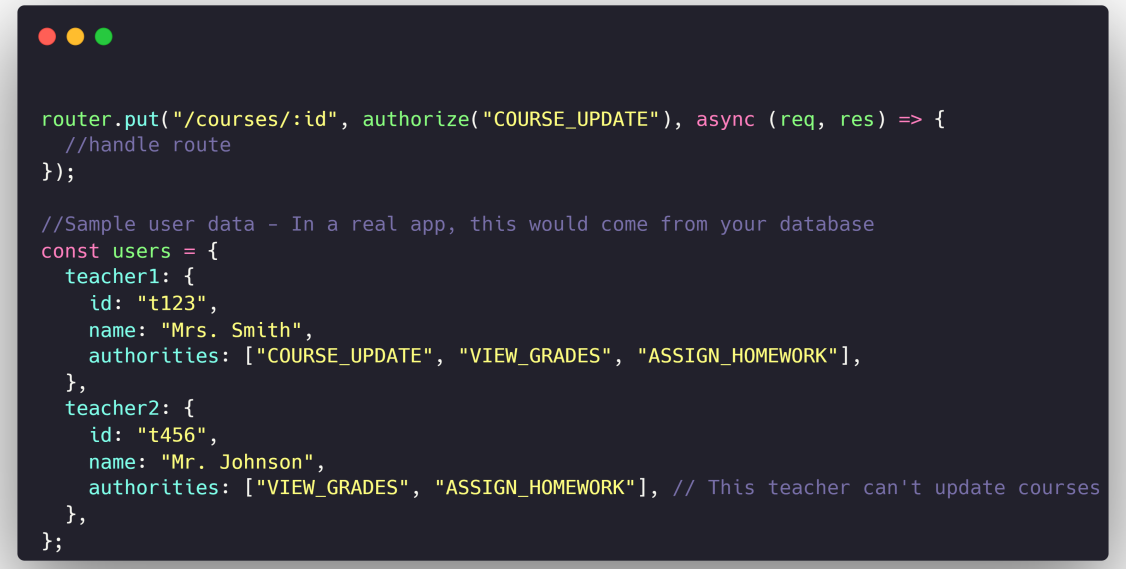
```
// Route: Deny update if the user is a student
app.put("/update-course/:id", (req, res) => {

    //handle route: get ID, Course and User

  if (user && user.role === "student") {
    return res.status(403).json({
      error: "Unauthorized: Students are not allowed to update courses",
    });
  }

  const updatedCourse = courseService.update(id, course);
  res.json(updatedCourse);
});
```

Cons: checking what is not allowed

```
// Route: Secure update, only Teachers and Admins are allowed
app.put("/secure-update-course/:id", (req, res) => {

    //handle route: get ID, User and Course

  if (user && (user.role === "teacher" || user.role === "admin")) {
    const updatedCourse = courseService.update(id, course);
    return res.json(updatedCourse);
  }

  res.status(403).json({
    error: "Unauthorized: Only Teachers and Admins are allowed to update courses",
  });
});
```

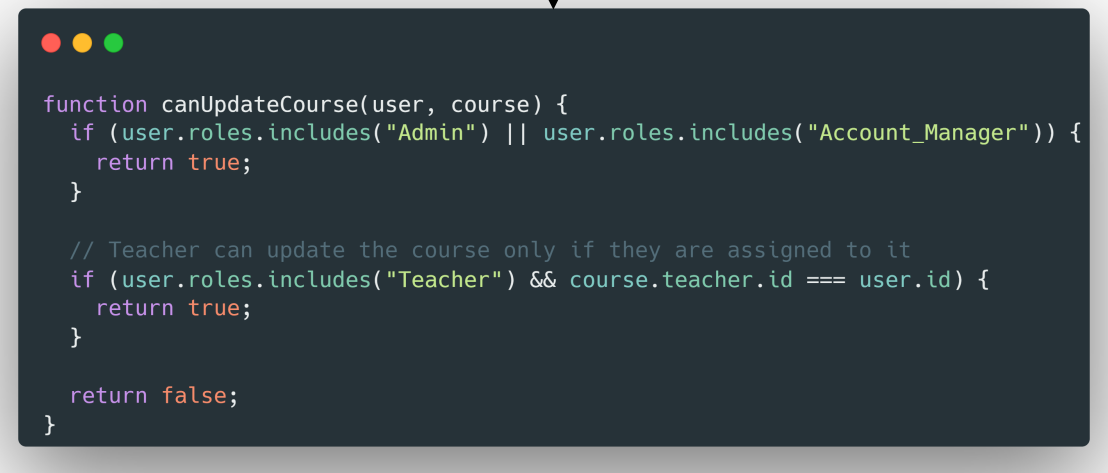
Cons: Role Explosion (for larger systems)
Pros: Deny by default

```
function hasPrivilege(user, requiredPrivilege) {
  if (!user || !user.authorities || !requiredPrivilege) return false;
  return user.authorities.includes(requiredPrivilege);
}

function authorize(requiredPrivilege) {
  return (req, res, next) => {
    const user = req.user; // Assuming req.user is set by previous authentication middleware

    if (!user || !hasPrivilege(user, requiredPrivilege)) {
      return res
        .status(403)
        .json({ error: "Unauthorized: Insufficient privileges" });
    }
    next(); // User has the required privilege, proceed to the next middleware/controller
  };
}

module.exports = { authorize };
```



```
router.put("/courses/:id", authorize("COURSE_UPDATE"), async (req, res) => {
  //handle route
});

//Sample user data - In a real app, this would come from your database
const users = {
  teacher1: {
    id: "t123",
    name: "Mrs. Smith",
    authorities: ["COURSE_UPDATE", "VIEW_GRADES", "ASSIGN_HOMEWORK"],
  },
  teacher2: {
    id: "t456",
    name: "Mr. Johnson",
    authorities: ["VIEW_GRADES", "ASSIGN_HOMEWORK"], // This teacher can't update courses
  },
};
```

Cons: Broken Object Level Auth. - Can update any course as long as you know ID?
Pros: Custom security expression, No hardcoding



```
function canUpdateCourse(user, course) {
  if (user.roles.includes("Admin") || user.roles.includes("Account_Manager")) {
    return true;
  }

  // Teacher can update the course only if they are assigned to it
  if (user.roles.includes("Teacher") && course.teacher.id === user.id) {
    return true;
  }

  return false;
}
```

```
router.put("/courses/:id", async (req, res) => {

  //handle route: get ID, User and Course

  if (canUpdateCourse(user, course)) {
    try {
      const updatedCourse = await courseService.update(id, course);
      return res.json(updatedCourse);
    } catch (error) {
      return res.status(500).json({ error: error.message });
    }
  }

  return res.status(403).json({
    error: "Unauthorized: You do not have permission to update this course",
  });
});
```

Cons: Database exploitation (no source of truth)
Pros: Least privilege, Deny by default

```
router.put("/courses/:id", async (req, res) => {
  //handle route: get ID, User and Course

  try {
    // Fetch course from the database
    const courseFromDB = await courseService.findById(id);
    if (!courseFromDB) {
      return res.status(404).json({ error: "Course not found" });
    }

    if (canUpdateCourse(user, course)) {
      try {
        const updatedCourse = await courseService.update(id, course);
        return res.json(updatedCourse);
      } catch (error) {
        return res.status(500).json({ error: error.message });
      }
    }

    return res.status(403).json({
      error: "Unauthorized: You do not have permission to update this course",
    });
  } catch (error) {
    return res.status(500).json({ error: error.message });
  }
});
```

Final Code (tally from the database first)

2. Property Level Issues (Data Control)

```
// Simulated user data from a database
const users = [
  { id: 1, name: "Alice", email: "alice@example.com", password: "hashed_password", isAdmin: true },
  { id: 2, name: "Bob", email: "bob@example.com", password: "hashed_password2", isAdmin: false }
];

// Endpoint that returns user data (without filtering sensitive info)
app.get("/users", (req, res) => {
  res.json(users);  // 🚨 Exposes passwords & isAdmin status!
});
```

Problems in the above code:

❌ Exposes passwords

❌ Reveals admin status

❌ Potential data misuse

```
// Simulated user data from a database
const users = [
  { id: 1, name: "Alice", email: "alice@example.com", password: "hashed_password", isAdmin: true },
  { id: 2, name: "Bob", email: "bob@example.com", password: "hashed_password2", isAdmin: false }
];

// DTO class to filter out sensitive fields
class UserDTO {
  constructor(user) {
    this.id = user.id;
    this.name = user.name;
    this.email = user.email;
  }
}

// Secure endpoint using DTO
app.get("/users", (req, res) => {
  const safeUsers = users.map(user => new UserDTO(user));  // ✅ Removes sensitive fields
  res.json(safeUsers);
});
```

Improvements:

✅ Passwords are no longer exposed

✅ Admin status is hidden, preventing privilege enumeration attacks

✅ Only necessary fields (id, name, email) are returned

✅ Prevents accidental data leaks in future updates

3. Input Validation

```
app.post("/register", (req, res) => {
  const { username, age, email, role } = req.body;

  // No validation
  if (!username || !age || !email || !role) {
    return res.status(400).json({ error: "All fields are required"
})}

  res.json({ message: "User registered successfully!" });
});
```

Issues:

❌No Input Validation – Allows any data type, length, or format.

❌No Role Restriction – Any string can be used as a role.

❌Potential DoS Attack Risk – Large or malformed inputs can overload the system.

```
const UserRole = Object.freeze({
    USER: "user",
    ADMIN: "admin",
    MODERATOR: "moderator",
});

// Validation schema
const registerSchema = Joi.object({
    username: Joi.string().trim().min(3).max(20).regex(/^[a-zA-Z0-9_]+$/).required(),
    age: Joi.number().integer().min(13).max(120).required(),
    email: Joi.string().trim().email().max(255).required(),
    role: Joi.string().valid(...Object.values(UserRole)).required() // Restrict roles to the enum values
});

app.post("/register", (req, res) => {
  const { error, value } = registerSchema.validate(req.body);

  if (error) {
    return res.status(400).json({ error: error.details[0].message });
  }

  // Create a new user instance
  const newUser = new User(value.username, value.age, value.email, value.role);

  res.json({ message: "User registered successfully!", user: newUser });
});
```

Improvements

✅ Enums for Role Validation: Prevents arbitrary values.

✅ DTO for Input Validation: Enforces strict validation rules.

✅ Model for Data Structuring: Ensures consistency.

3.1 Parameters Validation

```
// Insecure: No validation on the backend
app.get('/api/data', async (req, res) => {
  const { page, pageSize } = req.query;

  // Directly uses user input in database query
  const data = await db.query('SELECT * FROM records LIMIT ? OFFSET ?',
[   pageSize,
    (page - 1) * pageSize,
  ]);

  res.json(data);
});
```

Issues:

❌No Input Validation

❌Denial of Service (DoS) Risk

❌Excessive Data Exposure

```
// Secure: Backend validation (critical for security)
app.get("/api/data", async (req, res) => {
  const { page, pageSize } = req.query;

  // Validate inputs
  if ( !page || !pageSize || isNaN(page) || isNaN(pageSize) ||page < 1 ||
    pageSize < 1 || pageSize > 100 // Enforce strict maximum
  ) {
    return res.status(400).json({ error: "Invalid parameters" });
  }

  // Convert to integers explicitly
  const parsedPage = parseInt(page);
  const parsedPageSize = parseInt(pageSize);

  // Safely use validated values
  const data = await db.query("SELECT * FROM records LIMIT ? OFFSET ?", [
    parsedPageSize,
    (parsedPage - 1) * parsedPageSize,
  ]);

  res.json(data);
});
```

Improvements

✅ Validated and Sanitized inputs with parameterized queries.

✅ Type Safety

3.2 SQL Injection

```
const connection = mysql.createConnection({
  host: "localhost", user: "root", password: "", database: "users_db",
});

app.get("/user", (req, res) => {
  const username = req.query.username;

  // ❌ Vulnerable to SQL Injection
  const query = `SELECT * FROM users WHERE username = '${username}'`;

  connection.query(query, (error, results) => {
    if (error) {
      res.status(500).json({ error: "Database error" });
      return;
    }
    res.json(results);
  });
});
```

Injection Type-1:

```
/* If an attacker sends admin' -- as the username, the query becomes: */
SELECT * FROM users WHERE username = 'admin' -- '
/* the -- makes the rest of the query a comment, effectively bypassing authentication. */
```

Injection Type-2:

```
/* If an attacker enters admin' OR '1'='1, the query becomes */
SELECT * FROM users WHERE username = 'admin' OR '1'='1'
/* This condition is always true, allowing unauthorized access. */
```

```
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "",
  database: "users_db",
});

app.get("/user", (req, res) => {
  const username = req.query.username;

  // ✅ Secure against SQL Injection
  const query = `SELECT * FROM users WHERE username = ?`;

  connection.execute(query, [username], (error, results) => {
    if (error) {
      res.status(500).json({ error: "Database error" });
      return;
    }
    res.json(results);
  });
});
```

Final Code

- **Why is this secure?**
  - The ? placeholder ensures that user input is properly escaped and treated as data, not code.
  - The database handles binding the parameter, preventing malicious injections.
  - Even if an attacker tries admin' OR '1'='1, the query remains:

```
SELECT * FROM users WHERE username = 'admin\' OR \'1\'=\'1'
```

---

4. File Uploads

```
app.use(fileUpload());
const UPLOAD_DIR = path.join(__dirname, "uploads");

app.post("/upload", (req, res) => {
  if (!req.files || !req.files.file) {
    return res.status(400).send("No file uploaded.");
  }

  let file = req.files.file;
  let uploadPath = path.join(UPLOAD_DIR, file.name);
  // ❌ No validation, allows path traversal

  file.mv(uploadPath, (err) => {
    if (err) return res.status(500).send(err);
    res.send("File uploaded!");
  });
});
```

Security Issues in the Above Code

1. Path Traversal: A user can upload a file with a name like ../../evil.sh, escaping the upload directory.
2. No File Type Validation: Accepts any file type, including malicious ones.
3. No File Size Limit: Attackers can upload huge files to crash the server.
4. Trusting Content-Type Header: Easily spoofed by attackers.
5. No File Name Length Restriction: Very long filenames can cause issues.

```
app.use(
  fileUpload({ limits: { fileSize: 5 * 1024 * 1024 }}));
// ✅ Limit file size to 5MB

const UPLOAD_DIR = path.join(__dirname, "uploads");
const ALLOWED_EXTENSIONS = [".png", ".jpg", ".jpeg", ".pdf"]; // ✅ Allow only specific extensions
const MAX_FILENAME_LENGTH = 100; // ✅ Restrict file name length

// Ensure upload directory exists
if (!fs.existsSync(UPLOAD_DIR)) fs.mkdirSync(UPLOAD_DIR);

app.post("/upload", (req, res) => {
  if (!req.files || !req.files.file) return res.status(400).send("No file uploaded.");

  let file = req.files.file;
  // ✅ Prevent directory traversal by normalizing path
  let safeFileName = path.basename(file.name);

  // ✅ Validate file name length
  if (safeFileName.length > MAX_FILENAME_LENGTH) return res.status(400).send("File name is too long.")

  // ✅ Validate file extension
  let ext = path.extname(safeFileName).toLowerCase();
  if (!ALLOWED_EXTENSIONS.includes(ext)) return res.status(400).send("Invalid file type.")

  // ✅ Use a secure random filename to avoid overwriting or enumeration attacks
  let newFileName = `${Date.now()}-${Math.random().toString(36).substring(7)}${ext}`;
  let uploadPath = path.join(UPLOAD_DIR, newFileName);

  // ✅ Read file metadata directly (not trusting `req.files.file.mimetype`)
  const buffer = file.data;
  const fileSignature = buffer.toString("hex", 0, 4); // Check first few bytes for real type

  const ALLOWED_SIGNATURES = {
    ".png": "89504e47",
    ".jpg": ["ffd8ffe0", "ffd8ffe1", "ffd8ffe2", "ffd8ffe3", "ffd8ffe8"],
    ".jpeg": ["ffd8ffe0", "ffd8ffe1", "ffd8ffe2", "ffd8ffe3", "ffd8ffe8"],
    ".pdf": "25504446",
  };

  if (!Object.values(ALLOWED_SIGNATURES).includes(fileSignature)) {
    return res.status(400).send("File does not match expected format.");
  }

  // ✅ Move file securely
  file.mv(uploadPath, (err) => {
    if (err) return res.status(500).send("Error saving file.");
    res.send({ message: "File uploaded successfully.", fileName: newFileName });
  });
});
```

Final Code

5. CORS (cross origin resource sharing)

```
const cors = require("cors");

app.use(cors());
// Unsafe: Allow all origins and methods

app.get("/api/sensitive-data", (req, res) => {
  const userData = {
    userID: "12345",
    creditCard: "1234-5678-9012-3456",
    ssn: "123-45-6789",
  };
  res.json(userData);
});
```

❌Allows all origins- risking unauthorized access

❌Exposes highly data

❌Lacks authentication

```
// Define secure CORS options
const corsOptions = {
  // Only allow trusted domains
  origin: ["https://trusted-site.com", "https://app.trusted-site.com"],
  // Limit allowed HTTP methods to only those needed
  methods: ["GET"],
  // Restrict headers to only necessary ones
  allowedHeaders: ["Content-Type", "Authorization"],
  // Enable cookies and credentials from allowed origins
  credentials: true,
  // Improve preflight response for compatibility, legacy browsers (IE11)
  optionsSuccessStatus: 200,
};

app.use(cors(corsOptions));

app.get("/api/data", (req, res) => {
  // Check for an Authorization header before sending data
  if (!req.headers.authorization) {
    return res.status(401).json({ error: "Authentication required" });
  }

  // Return only non-sensitive data (can be done using a DTO in real-world example)
  res.json({
    userID: "12345",
    accountType: "premium",
  });
});
```

✅ Restricted Origins

✅ Limited Methods/Headers

✅ Authentication Check and Minimal Exposure

5 Open Source Security Tools Developers Should Know

S No.	Tool	Result Quality	DevX	Customizability	Maturity
1	**SemGrep OSS (Code scanner)**	• 2000+ rules ported from OSS tools (gitleaks, findsecbugs, gosec, more) • Supports over 30+ languages	• Runs everywhere (CLI, CI/CD, Docker, IDE) • Very fast: no compilation needed	• Very extensible, with many outputs formats • Anyone can write rules	Large Community of active contributors, many years of development
2	**OSV-Scanner (Dependency Checker)**	• Leverage OSV DB maintained by Google • Aggregates curated sources, i.e. Github Security Advisories • Supports 13 Languages	• Uses the OSV schema • Can run anywhere (local, IDE, CI)	• Ability to scan specific SBOM and lockfiles • Multiple options (ignore, recursive)	Growing popularity & Community
3	**KICS (IaC Scanner)**	• Includes 2000+ queries supporting 18 frameworks • Nightly Build • Curated rules with unit tests	• Provides 200+ build-in remediation recipes • Runs everywhere (IDE plugin, local, CI)	• Queries written in OPA (Rego) • Ability to support new frameworks	Growing popularity & community of contributors
4	**Trivy (Container Scanner)**	• Supports scanning container images, file systems, git repos, Virtual Machines, secrets, IaC(Kubernetes, Terraform) • Can generate SBOM	• Fast, no setup or prerequisites (i.e. Database or external libs) • Runs everywhere (IDE plugin, docker, local, CI)	• Extensible through modules (write your own detection logic) • Plugin to extend the CLI	• High popularity • Large community
5	**ZAP (Runtime Scanner)**	• Numerous features, detects the OWASP Top 10 risks • 250+ curated rules	• Runs everywhere (docker, desktop app) • Includes a headless mode to integrate in CI/CD pipeline	• Extensible through extensions (100+ available today) • Plugins to extend the CLI	• GitHub top 1000 project • Very popular & large community • ZAP Marketplace