Secure Coding Review

Let's choose **JavaScript (Node.js)** as the programming language and a **web application** built with **Express.js** as the application. We'll review the code for security vulnerabilities and provide recommendations for secure coding practices.

Example Node.js Web Application Code

Below is a simple Express.js application that allows users to log in and view their profile:

```
const express = require("express");
const session = require("express-session");
const bcrypt = require("bcrypt");
const app = express();

// Middleware for session management
app.use(
    session({
        secret: "supersecretkey", // Hardcoded secret key
        resave: false,
        saveUninitialized: true,
        cookie: { secure: false }, // Cookie not secure
    })
);

// Mock user database
const users = {
    admin: bcrypt.hashSync("admin123", 10),
    user: bcrypt.hashSync("user123", 10),
};

// Login route
app.post("/login", (req, res) => {
    const { username, password } = req.body;

    if (users[username] && bcrypt.compareSync(password, users[username])) {
        req.session.username = username;
        res.send("Login successful!");
    } else {
        res.status(401).send("Invalid credentials");
    }
});

// Profile route
app.get("/profile", (req, res) => {
    if (req.session.username) {
        res.send(`Welcome, ${req.session.username}!`);
    } else {
        res.redirect("/login");
    }
});

app.listen(3000, () => {
    console.log("Server running on http://localhost:3000");
});
```

Security Vulnerabilities and Recommendations

1. Hardcoded Secret Key

- **Issue**: The **secret** key for session management is hardcoded in the source code. This is insecure because it exposes the key to anyone with access to the code.
- **Recommendation**:
  - Store the secret key in environment variables.
  - Use a strong, randomly generated key.
  - Example:

```
const session = require("express-session");
app.use(
    session({
        secret: process.env.SECRET_KEY || "fallback_secret_key",
        resave: false,
        saveUninitialized: true,
        cookie: { secure: true, httpOnly: true },
    })
);
```

2. Insecure Session Cookie

- **Issue**: The session cookie is not marked as **secure** or **httpOnly**, making it vulnerable to theft via man-in-the-middle (MITM) attacks or client-side script access.
- **Recommendation**:
  - Set the **secure** flag to ensure cookies are only sent over HTTPS.
  - Set the **httpOnly** flag to prevent client-side script access.
  - Example:

```
cookie: { secure: true, httpOnly; true }
```

3. Lack of Input Validation

- **Issue**: The application does not validate or sanitize user inputs, which could lead to injection attacks or other vulnerabilities.
- **Recommendation**:
  - Use a library like **express-validator** to validate and sanitize user inputs.
  - Example:

```
const { body, validationResult } = require("express-validator");

app.post(
    "/login",
    [
        body("username").trim().escape(),
        body("password").trim().escape(),
    ],
    (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }

        const { username, password } = req.body;
        if (users[username] && bcrypt.compareSync(password, users[username])) {
            req.session.username = username;
            res.send("Login successful!");
        } else {
            res.status(401).send("Invalid credentials");
        }
    }
);
```

4. No HTTPS Enforcement

- **Issue**: The application does not enforce HTTPS, which could lead to man-in-the-middle (MITM) attacks.
- **Recommendation**:
  - Use a reverse proxy like Nginx or Apache to enforce HTTPS.
  - Use the **helmet** middleware to enforce security headers.
  - Example:

```
const helmet = require("helmet");
app.use(helmet());
```

5. No Rate Limiting

- **Issue**: The application does not implement rate limiting, making it vulnerable to brute-force attacks.
- **Recommendation**:
  - Use the **express-rate-limit** middleware to limit the number of login attempts.
  - Example:

```
const helmet = require("helmet");
app.use(helmet());

const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // Limit each IP to 5 requests per windowMs
});

app.use("/login", limiter);
```

6. No CSRF Protection

- **Issue**: The application does not protect against Cross-Site Request Forgery (CSRF) attacks.
- **Recommendation**:
  - Use the **csurf** middleware to add CSRF protection.
  - Example:

```
const csrf = require("csurf");
const csrfProtection = csrf({ cookie: true });

app.use(csrfProtection);

app.get("/login", (req, res) => {
    res.render("login", { csrfToken: req.csrfToken() });
});

app.post("/login", csrfProtection, (req, res) => {
    // Handle login
});
```

Tools for Static Code Analysis

To automate the process of identifying security vulnerabilities, you can use static code analysis tools. Some popular tools for Node.js include:

1. **ESLint**:
    - A linter for JavaScript that can identify potential security issues.
    - Install: **npm install eslint --save-dev**
    - Run: **npx eslint your_app_directory/**
2. **Snyk**:
    - Checks for known vulnerabilities in dependencies.
    - Install: **npm install -g snyk**
    - Run: **snyk test**

3. **Node Security Platform (nsp)**:
    - Identifies security vulnerabilities in dependencies.
    - Install: **npm install -g nsp**
    - Run: **nsp check**

Example ESLint Output

Running ESLint on the Node.js application might produce output like this:

```
/your_app_directory/app.js
  5:5  error  'session' is assigned a value but never used  no-unused-vars
 10:5  error  'bcrypt' is assigned a value but never used   no-unused-vars

✖ 2 problems (2 errors, 0 warnings)
```

Conclusion
By addressing the vulnerabilities identified above and following secure coding practices, you can significantly improve the security of your Node.js web application. Always use tools like ESLint, Snyk, and nsp to automate security checks and stay updated with the latest security best practices.