Security Scan Results

Generated using AI-powered vulnerability scanning with openai

Validated Vulnerabilities (Confidence "e 50%)

High Severity Vulnerability

SQL Injection

Function: executeQuery

Description:
The function directly interpolates user input into an SQL query without any sanitization or parameterization, making it vulnerable to SQL injection attacks.

Location:
Line 2

Vulnerable Code:
```
1: function executeQuery(query: string) {
2:   return `SELECT * FROM users WHERE id = ${query}`;
3: }
4:
5: function displayUserInput(input: string) {
6:   const element = document.createElement("div");
```

Remediation Steps:
Use parameterized queries or prepared statements to prevent SQL injection. Example:

```typescript
function executeQuery(query: string) {
  const sql = 'SELECT * FROM users WHERE id = ?';
  // Use a database library that supports parameterized queries
  database.execute(sql, [query]);
}
```

Confidence Score:
90.0%

High Severity Vulnerability

Cross-site Scripting (XSS)

Function: displayUserInput

Description:
The function sets `innerHTML` with user input directly, which can lead to XSS if the input is not properly sanitized.

Location:
Line 6

Vulnerable Code:
```
4:
5: function displayUserInput(input: string) {
6:   const element = document.createElement("div");
7:   element.innerHTML = input;
8:   document.body.appendChild(element);
9: }
10:
```

Remediation Steps:
Escape or sanitize user input before inserting it into the DOM. Example:

```typescript
function displayUserInput(input: string) {
  const element = document.createElement("div");
  element.textContent = input; // Use textContent to prevent XSS
  document.body.appendChild(element);
}
```

Confidence Score:
90.0%

High Severity Vulnerability

Command Injection

Function: processCommand

Description:
The function uses `exec` to run shell commands with user input, which can lead to command injection if the input is not properly validated.

Location:
Line 11

Vulnerable Code:
```
9: }
10:
11: function processCommand(cmd: string) {
12:   const exec = require("child_process").exec;
13:   exec(cmd, (error: any, stdout: any) => {
14:     console.log(stdout);
15:   });
```

Remediation Steps:
Validate and sanitize the input to ensure it does not contain malicious shell commands. Consider using `execFile` for safer execution. Example:

```typescript
function processCommand(cmd: string) {
  const execFile = require("child_process").execFile;
  execFile(cmd, (error: any, stdout: any) => {
    console.log(stdout);
  });
}
```

Confidence Score:
90.0%

High Severity Vulnerability

Cryptographic Failures

Function: storePassword

Description:
The function uses MD5 for hashing passwords, which is considered insecure due to its vulnerability to collision attacks.

Location:
Line 16

Vulnerable Code:
```
14:     console.log(stdout);
15:   });
16: }
17:
18: function storePassword(password: string) {
19:   const crypto = require("crypto");
20:   return crypto.createHash("md5").update(password).digest("hex");
```

Remediation Steps:
Use a stronger hashing algorithm like bcrypt or Argon2 for password hashing. Example:

```typescript
const bcrypt = require('bcrypt');
function storePassword(password: string) {
  const saltRounds = 10;
  return bcrypt.hashSync(password, saltRounds);
}
```

Confidence Score:
90.0%

Medium Severity Vulnerability

Race Condition

Function: withdrawMoney

Description:
The function checks the balance and then performs an asynchronous operation before deducting the amount, which can lead to race conditions if multiple withdrawals are attempted simultaneously.

Location:
Line 25

Vulnerable Code:
```
23: function transferFunds(amount: number, toAccount: string) {
24:   if (amount > 0) {
25:     return `Transferred ${amount} to ${toAccount}`;
26:   }
27: }
28:
29: let balance = 100;
```

Remediation Steps:
Use locks or transactions to ensure atomicity of the balance check and deduction. Example:

```typescript
async function withdrawMoney(amount: number) {
  await lock.acquire();
  try {
    if (balance >= amount) {
      balance -= amount;
      return amount;
    }
  } finally {
    lock.release();
  }
  return 0;
}
```

Confidence Score:
70.0%