OWASP NodeGoat Tutorial

Chetan Karande

Published with GitBook

Table of Contents

Welcome to nodegoat tutorial

A1- Injection

Description

Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

- Exploitability: EASY
- Prevalence: COMMON
- Detectability: AVERAGE
- Technical Impact: SEVERE

A1 - 1 Server Side JS Injection

Description

When `eval()`, `setTimeout()`, `setInterval()`, `Function()` are used to process user provided inputs, it can be exploited by an attacker to inject and execute malicious JavaScript code on server.

Attack Mechanics

Web applications using the JavaScript `eval()` function to parse the incoming data without any type of input validation are vulnerable to this attack. An attacker can inject arbitrary JavaScript code to be executed on the server. Similarly `setTimeout()`, and `setInterval()` functions can take code in string format as a first argument causing same issues as `eval()`.

This vulnerability can be very critical and damaging by allowing attacker to send various types of commands.

Denial of Service Attack:

> **Embedded Video: https://www.youtube.com/watch?v=krOx9QWwcYw**

An effective denial-of-service attack can be executed simply by sending the commands below to `eval()` function:

```
while(1)
```

This input will cause the target server's event loop to use 100% of its processor time and unable to process any other incoming requests until process is restarted.

An alternative DoS attack would be to simply exit or kill the running process:

```
process.exit()
```

or

```
process.kill(process.pid)
```

File System Access

Another potential goal of an attacker might be to read the contents of files from the server. For example, following two commands list the contents of the current directory and parent directory respectively:

```
res.end(require('fs').readdirSync('.').toString())

res.end(require('fs').readdirSync('..').toString())
```

Once file names are obtained, an attacker can issue the command below to view the actual contents of a file:

```
res.end(require('fs').readFileSync(filename))
```

An attacker can further exploit this vulnerability by writing and executing harmful binary files using `fs` and `child_process` modules.

How Do I Prevent It?

To prevent server-side js injection attacks:

- Validate user inputs on server side before processing
- Do not use `eval()` function to parse user inputs. Avoid using other commands with similar effect, such as `setTimeOut()`, `setInterval()`, and `Function()`.
- For parsing JSON input, instead of using `eval()`, use a safer alternative such as `JSON.parse()`. For type conversions use type related `parseXXX()` methods.
- Include `"use strict"` at the beginning of a function, which enables strict mode within the enclosing function scope.

Source Code Example

In `routes/contributions.js`, the `handleContributionsUpdate()` function insecurely uses `eval()` to covert user supplied contribution amounts to integer.

```
        // Insecure use of eval() to parse inputs
        var preTax = eval(req.body.preTax);
        var afterTax = eval(req.body.afterTax);
        var roth = eval(req.body.roth);
```

This makes application vulnerable to SSJS attack. It can fixed simply by using `parseInt()` instead.

```
        //Fix for A1 -1 SSJS Injection attacks - uses alternate method to eval
        var preTax = parseInt(req.body.preTax);
        var afterTax = parseInt(req.body.afterTax);
        var roth = parseInt(req.body.roth);
```

In addition, all functions begin with `use strict` pragma.

Further Reading

- “ServerSide JavaScript Injection: Attacking NoSQL and Node.js" a whitepaper by Bryan Sullivan.

A1 - 2 SQL and NoSQL Injection

Description

SQL and NoSQL injections enable an attacker to inject code into the query that would be executed by the database. These flaws are introduced when software developers create dynamic database queries that include user supplied input.

Attack Mechanics

Both SQL and NoSQL databases are vulnerable to injection attack. Here is an example of equivalent attack in both cases, where attacker manages to retrieve admin user's record without knowing password:

1. SQL Injection

Lets consider an example SQL statement used to authenticate the user with username and password

```
SELECT * FROM accounts WHERE username = '$username' AND password = '$password'
```

If this statement is not prepared or properly handled when constructed, an attacker may be able to supply `admin' --` in the username field to access the admin user's account bypassing the condition that checks for the password. The resultant SQL query would looks like:

```
SELECT * FROM accounts WHERE username = 'admin' -- AND password = ''
```

2. NoSQL Injection

The equivalent of above query for NoSQL MongoDB database is:

```
db.accounts.find({username: username, password: password});
```

While here we are no longer dealing with query language, an attacker can still achieve the same results as SQL injection by supplying JSON input object as below:

```
{
    "username": "admin",
    "password": {$gt: ""}
}
```

In MongoDB, `$gt` selects those documents where the value of the field is greater than (i.e. >) the specified value. Thus above statement compares password in database with empty string for greatness, which returns `true`.

The same results can be achieved using other comparison operator such as `$ne`.

How Do I Prevent It?

Here are some measures to prevent SQL / NoSQL injection attacks, or minimize impact if it happens:

- Prepared Statements: For SQL calls, use prepared statements instead of building dynamic queries using string concatenation.
- Input Validation: Validate inputs to detect malicious values. For NoSQL databases, also validate input types against expected types
- Least Privilege: To minimize the potential damage of a successful injection attack, do not assign DBA or admin type access rights to your application accounts. Similarly minimize the privileges of the operating system account that the database process runs under.

A2-Broken Authentication and Session Management

Description

In this attack, an attacker (who can be anonymous external attacker, a user with own account who may attempt to steal data from accounts, or an insider wanting to disguise his or her actions) uses leaks or flaws in the authentication or session management functions to impersonate other users. Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.

- Exploitability: AVERAGE
- Prevalence: WIDESPREAD
- Detectability: AVERAGE
- Technical Impact: SEVERE

A2 - 1 Session Management

Description

Session management is a critical piece of application security. It is broader risk, and requires developers take care of protecting session id, user credential secure storage, session duration, and protecting critical session data in transit.

Attack Mechanics

Scenario #1: Application timeouts aren't set properly. User uses a public computer to access site. Instead of selecting “logout” the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #2: Attacker acts as a man-in-middle and acquires user's session id from network traffic. Then uses this authenticated session id to connect to application without needing to enter user name and password.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.

How Do I Prevent It?

Session management related security issues can be prevented by taking these measures:

- User authentication credentials should be protected when stored using hashing or encryption.
- Session IDs should not be exposed in the URL (e.g., URL rewriting).
- Session IDs should timeout. User sessions or authentication tokens should get properly invalidated during logout.
- Session IDs should be recreated after successful login.
- Passwords, session IDs, and other credentials should not be sent over unencrypted connections.

Source Code Examples

In the insecure demo app, following issues exists:

1. Protecting user credentials

password gets stored in database in plain text . Here is related code in `data /user-dao.js` `addUser()` method:

```
// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: password //received from request param
};
```

To secure it, handle password storage in a safer way by using one way encryption using salt hashing as below:

```
// Generate password hash
var salt = bcrypt.genSaltSync();
var passwordHash = bcrypt.hashSync(password, salt);

// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: passwordHash
};
```

This hash password can not be decrypted, hence more secure. To compare the password when user logs in, the user entered password gets converted to hash and compared with the hash in storage.

```
if (bcrypt.compareSync(password, user.password)) {
    callback(null, user);
} else {
    callback(invalidPasswordError, null);
}
```

Note: The bcrypt module also provides asynchronous methods for creating and comparing hash.

2. Session timeout and protecting cookies in transit

The insecure demo application does not contain any provision to timeout user session. The session stays active until user explicitly logs out.

In addition to that, the app does not prevent cookies being accessed in script, making application vulnerable to Cross Site Scripting (XSS) attacks. Also cookies are not prevented to get sent on insecure HTTP connection.

To secure the application:

1. Use session based timeouts, terminate session when browser closes.

```
// Enable session management using express middleware
app.use(express.cookieParser());
```

1. In addition, sets `HTTPOnly` HTTP header preventing cookies being accessed by scripts. The application used HTTPS secure connections, and cookies are configured to be sent only on Secure HTTPS connections by setting `Secure` flag.

```
app.use(express.session({
    secret: "s3Cur3",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

1. When user clicks logout, destroy the session and session cookie

```
req.session.destroy(function() {
    res.redirect("/");
});
```

Note: The example code uses `MemoryStore` to manage session data, which is not designed for production environment, as it will leak memory, and will not scale past a single process. Use database based storage MongoStore or RedisStore for production. Alternatively, sessions can be managed using popular passport module.

Further Reading

- Helmet Security header middleware collection for express
- Seven Web Server HTTP Headers that Improve Web Application Security for Free
- Passport authentication middleware
- CWE-384: Session Fixation

A2 - 2 Password Guessing Attacks

Description

Implementing a robust minimum password criteria (minimum length and complexity) can make it difficult for attacker to guess password.

Attack Mechanics

The attacker can exploit this vulnerability by brute force password guessing, more likely using tools that generate random passwords.

How Do I Prevent It?

Password length

Minimum passwords length should be at least eight (8) characters long. Combining this length with complexity makes a password difficult to guess and/or brute force.

Password complexity

Password characters should be a combination of alphanumeric characters. Alphanumeric characters consist of letters, numbers, punctuation marks, mathematical and other conventional symbols.

Username/Password Enumeration

Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code

Additional Measures

- For additional protection against brute forcing, enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed.
- Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly

identical in both display and source code
- Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception. Enforce the changing of temporary passwords on the next use. Temporary passwords and links should have a short expiration time.

Source Code Example

The demo application doesn't enforce strong password. In routes/session.js `validateSignup()` method, the regex for password enforcement is simply

```
var PASS_RE = /^.{1,20}$/;
```

A stronger password can be enforced using the regex below, which requires at least 8 character password with numbers and both lowercase and uppercase letters.

```
var PASS_RE =/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;
```

Another issue, in routes/session.js, the `handleLoginRequest()` enumerated whether password was incorrect or user doesn't exist.This information can be valuable to an attacker with brute forcing attempts. This can be easily fixed using a generic error message such as "Invalid username and/or password".