A1 - Injection

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

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.

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:

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 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode)within the enclosing function scope.

Source Code Example

In `routes/contributions.js` , the `handleContributionsUpdate()` function insecurely uses `eval()` to convert 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" (https://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf) 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` .

The demo application is vulnerable to the NoSQL Injection. For example, on the Allocations page, running a search with a malicious input `1'; return 1 == '1` retrieves allocations for all the users in the database.

SSJS Attack Mechanics

Server-side JavaScript Injection (SSJS) is an attack where JavaScript code is injected and executed in a server component. MongoDB specifically, is vulnerable to this attack when queries are run without proper sanitization.

$where operator

MongoDB's `$where` operator performs JavaScript expression evaluation on the MongoDB server. If the user is able to inject direct code into such queries then such an attack can take place

Lets consider an example query:

```
 db.allocationsCollection.find({ $where: "this.userId == '" + parsedUserId + "' && "
+ "this.stocks > " + "'" + threshold + "'" });
```

The code will match all documents which have a `userId` field as specified by `parsedUserId` and a `stocks` field as specified by `threshold`. The problem is that these parameters are not validated, filtered, or sanitised, and vulnerable to SSJS Injection.

NoSQL SSJS Injection

An attacker can send the following input for the `threshold` field in the requests query, which will create a valid JavaScript expression and satisfy the `$where` query as well, resulting in a DoS attack on the MongoDB server:

```
http://localhost:4000/allocations/2?threshold=5';while(true){};'
```

You can also just drop the following into the Stocks Threshold input box:

```
';while(true){};'
```

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.

For the above NoSQL vulnerability, bare minimum fixes can be found in `allocations.html` and `allocations-dao.js`

A1 - 3 Log Injection

Description

Log injection vulnerabilities enable an attacker to forge and tamper with an application's logs.

Attack Mechanics

An attacker may craft a malicious request that may deliberately fail, which the application will log, and when attacker's user input is unsanitized, the payload is sent as-is to the logging facility. Vulnerabilities may vary depending on the logging facility:

1. Log Forging (CRLF)

Lets consider an example where an application logs a failed attempt to login to the system. A very common example for this is as follows:

```
var userName = req.body.userName;
console.log('Error: attempt to login with invalid user: ', userName);
```

When user input is unsanitized and the output mechanism is an ordinary terminal stdout facility then the application will be vulnerable to CRLF injection, where an attacker can create a malicious payload as follows:

```
curl http://localhost:4000/login -X POST --data 'userName=vyva%0aError: alex moldovan
failed $1,000,000 transaction&password=Admin_123&_csrf='
```

Where the `userName` parameter is encoding in the request the LF symbol which will result in a new line to begin. Resulting log output will look as follows:

```
Error: attempt to login with invalid user:  vyva
Error: alex moldovan failed $1,000,000 transaction
```

2. Log Injection Escalation

An attacker may craft malicious input in hope of an escalated attack where the target isn't the logs themselves, but rather the actual logging system. For example, if an application has a back-office web app that manages viewing and tracking the logs, then an attacker may send an XSS payload into the log, which may not result in log forging on the log itself, but when viewed by a system administrator on the log viewing web app then it may compromise it and result in XSS injection that if the logs app is vulnerable.

How Do I Prevent It?

As always when dealing with user input:

- Do not allow user input into logs
- Encode to proper context, or sanitize user input

Encoding example:

```
// Step 1: Require a module that supports encoding
var ESAPI = require('node-esapi');
// - Step 2: Encode the user input that will be logged in the correct context
// following are a few examples:
console.log('Error: attempt to login with invalid user: %s', ESAPI.encoder().enco
deForHTML(userName));
console.log('Error: attempt to login with invalid user: %s', ESAPI.encoder().enco
deForJavaScript(userName));
console.log('Error: attempt to login with invalid user: %s', ESAPI.encoder().enco
deForURL(userName));
```

For the above Log Injection vulnerability, example and fix can be found at `routes/session.js`