Defensive Coding In JavaScript

Dangers of `eval`

The `eval()` function in JavaScript takes a string as an argument and executes it as Javascript source code. Not only is it slow to execute, but bad actors can also inject malicious code into the input string for mischievous reasons. Thus, it's best never to use it. If you MUST use it, only allow trusted and predetermined input through it. NEVER trust user input.

The functions, `setInterval()`, `setTimeout()`, and `new Function()` use `eval()` in their implementations, and should be used with the same caution.

```
// This user input causes an infinite loop to run
const user_input = "while(true) ;";
eval(user_input);

// This user input closes the application
const user_input = "process.exit(0)";
eval(user_input);
```

Dangers of `fs` Module

The `fs` module coupled with improperly sanitized user input gives attackers access to our entire file system and exposes it to vulnerabilities. To mitigate the risk, we can tweak our code to restrict traversal scope to a directory of our choice using `path.join()` and `process.cwd()`.

```
const user_input = "/system_file.cfg";
fs.unlinkSync(user_input); // Deletes important file

// Hard-code path to restrict scope
const root_directory = process.cwd();
const filePath = path.join(root_directory , fileName);
fs.unlinkSync(filePath); // File not found error
```

Dangers of Regular Expressions

Attackers can make use of insecure regex expressions to trigger a Regular expression Denial of Service (ReDoS). The RegEx engine can lead to catastrophic backtracking by taking an exponential amount of backtracking steps on poorly defined Regex expressions. To prevent this danger, we can use the validator npm package, which provides a library of string validators and sanitizers for things like IP addresses, emails, and phone numbers. We can also use tools like the safe-regex npm package to detect dangerous regular expressions.

String	Number of Digits	Number of Steps
123#	3	6
123456789123456789…#	180	6
1c#	1	5
1234567o#	7	755
123456789123456d#	15	196587
1234567891234567e#	16	TIMEOUT ERROR

JavaScript Strict Mode

JavaScript strict mode is a defensive tool that can reveal vulnerabilities in JavaScript code by throwing errors that would otherwise be silent. By intentionally enforcing different semantics, it will throw errors on things like assignments to undefined variables, duplicate parameters, deleting variables or functions, et cetera. To enable strict mode, simply add `"use strict";` to the beginning of the Javascript file.

```
// Runs fine without strict mode
x = "codecademy";

// Throws “ReferenceError” with strict mode
"use strict";
x = "codecademy";

// Runs fine with strict mode if variable is declared with
let, var, or const
"use strict";
var x = "codecademy";
```

Static Code Analysis

A lint, or linter, is a static code analysis tool used to evaluate and improve source code without executing it. It can find and flag programming errors, bugs, and patterns that may compromise security. The most popular JavaScript linters are ESLint, JSLint, and JSHint. They can be customized to one's needs by using configuration files or third-party plugins.

eslint-plugin-security is a plugin for ESlint that adds rules to detect several security vulnerabilities including unsafe regular expressions, non-literal `exec()`, `eval()` used with an expression, and more!

Dangers and Alternatives of `exec`

The `exec()` method can lead to a vulnerability where user input can run as a shell command. The danger is that unrestricted commands can access, modify, and delete files. The `execFile()` method is an alternative that works similarly to `exec()` but requires the separation of the commands and their arguments.

```
// Spawns a shell with the input as is
exec("ls -lah /tmp");

// Requires a command and specified arguments to execute
execFile("ls", ["-lah", "/tmp"]);
```

Print