Reporte de Reglas de SonarQube

Tabla de Contenido

Resumen de Reglas por Lenguaje

Lenguaje	Total Reglas	Reglas Activas	Reglas Depreciadas
JS	407	377	30

JS

Lines should not end with trailing whitespaces

Clave: javascript:S1131

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

Trailing whitespaces bring no information, they may generate noise when comparing different versions of the same file, and they can create bugs when they appear after a \ marking a line continuation. They should be systematically removed.

An automated code formatter allows to completely avoid this family of issues and should be used wherever possible.

```
// The following string will error if there is a whitespace after '\'
var str = "Hello \
World";
```

introduction

This rule is deprecated, and will eventually be removed.

Track uses of "FIXME" tags

Clave: javascript:S1134

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- CWE - CWE-546 - Suspicious Comment

root_cause

`FIXME` tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later.

Sometimes the developer will not have the time or will simply forget to get back to that tag.

This rule is meant to track those tags and to ensure that they do not go unnoticed.

```
function divide(numerator, denominator) {
  return numerator / denominator;              // FIXME denominator value might be  0
}
```

Track uses of "TODO" tags

Clave: javascript:S1135

Severidad: INFO

Estado: ✅ Activa

Descripción: No disponible

resources

- CWE - CWE-546 - Suspicious Comment

root_cause

Developers often use `TODO` tags to mark areas in the code where additional work or improvements are needed but are not implemented immediately. However, these `TODO` tags sometimes get overlooked or forgotten, leading to incomplete or unfinished code. This rule aims to identify and address unattended `TODO` tags to ensure a clean and maintainable codebase. This description explores why this is a problem and how it can be fixed to improve the overall code quality.

What is the potential impact?

Unattended `TODO` tags in code can have significant implications for the development process and the overall codebase.

Incomplete Functionality: When developers leave `TODO` tags without implementing the corresponding code, it results in incomplete functionality within the software. This can lead to unexpected behavior or missing features, adversely affecting the end-user experience.

Missed Bug Fixes: If developers do not promptly address `TODO` tags, they might overlook critical bug fixes and security updates. Delayed bug fixes can result in more severe issues and increase the effort required to resolve them later.

Impact on Collaboration: In team-based development environments, unattended `TODO` tags can hinder collaboration. Other team members might not be aware of the intended changes, leading to conflicts or redundant efforts in the codebase.

Codebase Bloat: The accumulation of unattended `TODO` tags over time can clutter the codebase and make it difficult to distinguish between work in progress and completed code. This bloat can make it challenging to maintain an organized and efficient codebase.

Addressing this code smell is essential to ensure a maintainable, readable, reliable codebase and promote effective collaboration among developers.

Noncompliant code example

```
function doSomething() {
  // TODO
}
```

Boolean expressions should not be gratuitous

Clave: javascript:S2589

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

introduction

Gratuitous boolean expressions are conditions that do not change the evaluation of a program. This issue can indicate logical errors and affect the correctness of an application, as well as its maintainability.

root_cause

Control flow constructs like `if`-statements allow the programmer to direct the flow of a program depending on a boolean expression. However, if the condition is always true or always false, only one of the branches will ever be executed. In that case, the control flow construct and the condition no longer serve a purpose; they become *gratuitous*.

What is the potential impact?

The presence of gratuitous conditions can indicate a logical error. For example, the programmer *intended* to have the program branch into different paths but made a mistake when formulating the branching condition. In this case, this issue might result in a bug and thus affect the reliability of the application. For instance, it might lead to the computation of incorrect results.

Additionally, gratuitous conditions and control flow constructs introduce unnecessary complexity. The source code becomes harder to understand, and thus, the application becomes more difficult to maintain.

resources

Articles & blog posts

- CWE - CWE-571 - Expression is Always True
- CWE - CWE-570 - Expression is Always False

how_to_fix

Gratuitous boolean expressions are suspicious and should be carefully removed from the code.

First, the boolean expression in question should be closely inspected for logical errors. If a mistake was made, it can be corrected so the condition is no longer gratuitous.

If it becomes apparent that the condition is actually unnecessary, it can be removed. The associated control flow construct (e.g., the `if`-statement containing the condition) will be adapted or even removed, leaving only the necessary branches.

Noncompliant code example

```
if (a) {
  if (a) { // Noncompliant
    doSomething();
  }
}
```

Compliant solution

```
if (a) {
  if (b) {
    doSomething();
  }
}

// or
if (a) {
  doSomething();
}
```

Default export names and file names should match

Clave: javascript:S3317

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

By convention, a file that exports only one class, function, or constant should be named for that class, function or constant. Anything else may confuse maintainers.

Noncompliant code example

```
// file path: myclass.js  -- Noncompliant
class MyClass {
  // ...
}
export default MyClass;
```

Compliant solution

```
// file path: MyClass.js
class MyClass {
  // ...
}
export default MyClass;
```

Exceptions

Case, dots (`.`), underscores (`_`) and dashes (`-`) are ignored from the name comparison. Postfixes in filenames like `.dev` in `my.class.dev.js` are also ignored.

Callbacks of array methods should have return statements

Clave: javascript:S3796

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Array

root_cause

In JavaScript, many array methods take a callback function as an argument. These methods are designed to transform or filter arrays based on the logic provided in the callback function. The callback function is called sequentially, and the return value of the callback function is used to determine the return value of the `Array` method.

If the callback function does not return a value, the array method may not work as expected and is most likely a mistake.

This rule applies to the following methods of an array:

- `Array.from`
- `Array.prototype.every`
- `Array.prototype.filter`
- `Array.prototype.find`
- `Array.prototype.findLast`
- `Array.prototype.findIndex`
- `Array.prototype.findLastIndex`
- `Array.prototype.map`
- `Array.prototype.flatMap`
- `Array.prototype.reduce`
- `Array.prototype.reduceRight`
- `Array.prototype.some`
- `Array.prototype.sort`
- `Array.prototype.toSorted`

If there is no `return`, the callback will implicitly return `undefined`, which may cause unexpected behavior or errors in the code.

```
let arr = ["a", "b", "c"];
let merged = arr.reduce(function(a, b) {
  a.concat(b); // Noncompliant: No return statement, will result in TypeError
});
```

Always add a return statement to the callback function passed to the array method.

```
let arr = ["a", "b", "c"];
let merged = arr.reduce(function(a, b) {
  return a.concat(b);
});
```

Variables and functions should not be declared in the global scope

Clave: javascript:S3798

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

Any variable or function declared in the global scope implicitly becomes attached to the global object (the `window` object in a browser environment). To make it explicit this variable or function should be a property of `window`. When it is meant to be used just locally, it should be declared with the `const` or `let` keywords (since ECMAScript 2015) or within an Immediately-Invoked Function Expression (IIFE).

This rule should not be activated when modules are used.

Noncompliant code example

```
var myVar = 42;       // Noncompliant
function myFunc() { } // Noncompliant
```

Compliant solution

```
window.myVar = 42;
window.myFunc = function() { };
```

or

```
let myVar = 42;
let myFunc = function() { }
```

or

```
// IIFE
(function() {
  var myVar = 42;
  function myFunc() { }
})();
```

Destructuring patterns should not be empty

Clave: javascript:S3799

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Destructuring assignment

root_cause

Destructuring is a convenient way of extracting multiple values from data stored in (possibly nested) objects and arrays. It can make code more concise and expressive by directly extracting values or properties needed from arrays or objects. However, it is possible to define an empty pattern that has no effect, where no variables are bound to the destructured values.

```
let {a: {}} = myObj; // Noncompliant: this does not create any variable
function foo({p: []}) { // Noncompliant: this does not define any parameter
  // ...
}
```

When empty curly or square brackets are bound to a pattern with a colon (`:`), like `{ pattern: [] }` or `{ pattern: {} }`, the intent is likely to define a default value. To properly define such a default value, use the assignment operator (`=`) instead.

```
let {a = {}} = myObj;
function foo({p = []}) {
  // ...
}
```

If that is not the intention, complete the destructuring pattern to contain the variables to create.

```
let {a: {b, c}} = myObj;
function foo({p: [a, b, c]}) {
  // ...
}
```

"default" clauses should be last

Clave: javascript:S4524

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `switch`

root_cause

`switch` can contain a `default` clause for various reasons: to handle unexpected values, to show that all the cases were properly considered, etc.

For readability purposes, to help a developer quickly spot the default behavior of a `switch` statement, it is recommended to put the `default` clause at the beginning or the end of the `switch` statement.

This rule raises an issue if the `default` clause is not the first or the last one of the `switch`'s cases.

```
switch (param) {
  case 0:
    doSomething();
    break;
  default: // Noncompliant: default clause should be the first or last one
    error();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

Allowing mixed-content is security-sensitive

Clave: javascript:S5730

Severidad: MINOR

Estado: ☑ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

Implement content security policy *block-all-mixed-content* directive which is supported by all modern browsers and will block loading of mixed-contents.

Compliant Solution

In Express.js application a standard way to block mixed-content is to put in place the helmet-csp or helmet middleware with the `blockAllMixedContent` directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      "default-src": ["'self'", 'example.com', 'code.jquery.com'],
      blockAllMixedContent: [] // Compliant
    }
  })
);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Mixed-content
- developer.mozilla.org - Content Security Policy (CSP)
- w3.org - Content Security Policy Level 3

default

A mixed-content is when a resource is loaded with the HTTP protocol, from a website accessed with the HTTPs protocol, thus mixed-content are not encrypted and exposed to MITM attacks and could break the entire level of protection that was desired by implementing encryption with the HTTPs protocol.

The main threat with mixed-content is not only the confidentiality of resources but the whole website integrity:

- A passive mixed-content (eg: *<img src="http://example.com/picture.png"*>) allows an attacker to access and replace only these resources, like images, with malicious ones that could lead to successful phishing attacks.
- With active mixed-content (eg: *<script src="http://example.com/library.js"*>) an attacker can compromise the entire website by injecting malicious javascript code for example (accessing and modifying the DOM, steal cookies, etc).

Ask Yourself Whether

- The HTTPS protocol is in place and external resources are fetched from the website pages.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement content security policy *block-all-mixed-content* directive which is supported by all modern browsers and will block loading of mixed-contents.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet-csp or helmet middleware is used without the `blockAllMixedContent` directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      "default-src": ["'self'", 'example.com', 'code.jquery.com']
    } // Sensitive: blockAllMixedContent directive is missing
  })
);
```

Compliant Solution

In Express.js application a standard way to block mixed-content is to put in place the helmet-csp or helmet middleware with the `blockAllMixedContent` directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      "default-src": ["'self'", 'example.com', 'code.jquery.com'],
      blockAllMixedContent: [] // Compliant
    }
  })
);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Mixed-content
- developer.mozilla.org - Content Security Policy (CSP)
- w3.org - Content Security Policy Level 3

assess_the_problem

Ask Yourself Whether

- The HTTPS protocol is in place and external resources are fetched from the website pages.

There is a risk if you answered yes to this question.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet-csp or helmet middleware is used without the `blockAllMixedContent` directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      "default-src": ["'self'", 'example.com', 'code.jquery.com']
    } // Sensitive: blockAllMixedContent directive is missing
  })
);
```

root_cause

A mixed-content is when a resource is loaded with the HTTP protocol, from a website accessed with the HTTPs protocol, thus mixed-content are not encrypted and exposed to MITM attacks and could break the entire level of protection that was desired by implementing encryption with the HTTPs protocol.

The main threat with mixed-content is not only the confidentiality of resources but the whole website integrity:

- A passive mixed-content (eg: *<img src="http://example.com/picture.png"*>) allows an attacker to access and replace only these resources, like images, with malicious ones that could lead to successful phishing attacks.
- With active mixed-content (eg: *<script src="http://example.com/library.js"*>) an attacker can compromise the entire website by injecting malicious javascript code for example (accessing and modifying the DOM, steal cookies, etc).

Disabling content security policy frame-ancestors directive is security-sensitive

Clave: javascript:S5732

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Clickjacking attacks occur when an attacker try to trick an user to click on certain buttons/links of a legit website. This attack can take place with malicious HTML frames well hidden in an attacker website.

For instance, suppose a safe and authentic page of a social network (https://socialnetworkexample.com/makemyprofilpublic) which allows an user to change the visibility of his profile by clicking on a button. This is a critical feature with high privacy concerns. Users are generally well informed on the social network of the consequences of this action. An attacker can trick users, without their consent, to do this action with the below embedded code added on a malicious website:

```
<html>
<b>Click on the button below to win 5000$</b>
<br>
<iframe src="https://socialnetworkexample.com/makemyprofilpublic" width="200" height="200"></iframe>
</html>
```

Playing with the size of the iframe it’s sometimes possible to display only the critical parts of a page, in this case the button of the *makemyprofilpublic* page.

assess_the_problem

Ask Yourself Whether

- Critical actions of the application are prone to clickjacking attacks because a simple click on a link or a button can trigger them.

There is a risk if you answered yes to this question.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet-csp or helmet middleware is used without the `frameAncestors` directive (or if `frameAncestors` is set to `'none'`):

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      // other directives
      frameAncestors: ["'none'"] // Sensitive: frameAncestors  is set to none
    }
  })
);
```

default

Clickjacking attacks occur when an attacker try to trick an user to click on certain buttons/links of a legit website. This attack can take place with malicious HTML frames well hidden in an attacker website.

For instance, suppose a safe and authentic page of a social network (https://socialnetworkexample.com/makemyprofilpublic) which allows an user to change the visibility of his profile by clicking on a button. This is a critical feature with high privacy concerns. Users are generally well informed on the social network of the consequences of this action. An attacker can trick users, without their consent, to do this action with the below embedded code added on a malicious website:

```
<html>
<b>Click on the button below to win 5000$</b>
<br>
<iframe src="https://socialnetworkexample.com/makemyprofilpublic" width="200" height="200"></iframe>
</html>
```

Playing with the size of the iframe it’s sometimes possible to display only the critical parts of a page, in this case the button of the *makemyprofilpublic* page.

Ask Yourself Whether

- Critical actions of the application are prone to clickjacking attacks because a simple click on a link or a button can trigger them.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement content security policy *frame-ancestors* directive which is supported by all modern browsers and will specify the origins of frame allowed to be loaded by the browser (this directive deprecates X-Frame-Options).

Sensitive Code Example

In Express.js application the code is sensitive if the helmet-csp or helmet middleware is used without the `frameAncestors` directive (or if `frameAncestors` is set to `'none'`):

```
const express = require('express');
const helmet = require('helmet');
```

```
let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      // other directives
      frameAncestors: ["'none'"] // Sensitive: frameAncestors  is set to none
    }
  })
);
```

Compliant Solution

In Express.js application a standard way to implement CSP frame-ancestors directive is the helmet-csp or helmet middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      // other directives
      frameAncestors: ["'example.com'"] // Compliant
    }
  })
);
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP Cheat Sheets - Clickjacking Defense Cheat Sheet
- developer.mozilla.org - Frame-ancestors
- developer.mozilla.org - Content Security Policy (CSP)
- CWE - CWE-451 - User Interface (UI) Misrepresentation of Critical Information
- w3.org - Content Security Policy Level 3

how_to_fix

Recommended Secure Coding Practices

Implement content security policy *frame-ancestors* directive which is supported by all modern browsers and will specify the origins of frame allowed to be loaded by the browser (this directive deprecates X-Frame-Options).

Compliant Solution

In Express.js application a standard way to implement CSP frame-ancestors directive is the helmet-csp or helmet middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      // other directives
      frameAncestors: ["'example.com'"] // Compliant
    }
  })
);
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP Cheat Sheets - Clickjacking Defense Cheat Sheet
- developer.mozilla.org - Frame-ancestors
- developer.mozilla.org - Content Security Policy (CSP)
- CWE - CWE-451 - User Interface (UI) Misrepresentation of Critical Information
- w3.org - Content Security Policy Level 3

Allowing browsers to sniff MIME types is security-sensitive

Clave: javascript:S5734

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- Content-Type header is not systematically set for all resources.
- Content of resources can be controlled by users.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

In Express.js application the code is sensitive if, when using helmet, the `noSniff` middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet({
    noSniff: false, // Sensitive
  })
);
```

default

MIME confusion attacks occur when an attacker successfully tricks a web-browser to interpret a resource as a different type than the one expected. To correctly interpret a resource (script, image, stylesheet …) web browsers look for the Content-Type header defined in the HTTP response received from the server, but often this header is not set or is set with an incorrect value. To avoid content-type mismatch and to provide the best user experience, web browsers try to deduce the right content-type, generally by inspecting the content of the resources (the first bytes). This "guess mechanism" is called MIME type sniffing.

Attackers can take advantage of this feature when a website ("example.com" here) allows to upload arbitrary files. In that case, an attacker can upload a malicious image *fakeimage.png* (containing malicious JavaScript code or a polyglot content file) such as:

```
<script>alert(document.cookie)</script>
```

When the victim will visit the website showing the uploaded image, the malicious script embedded into the image will be executed by web browsers performing MIME type sniffing.

Ask Yourself Whether

- Content-Type header is not systematically set for all resources.
- Content of resources can be controlled by users.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Implement X-Content-Type-Options header with *nosniff* value (the only existing value for this header) which is supported by all modern browsers and will prevent browsers from performing MIME type sniffing, so that in case of Content-Type header mismatch, the resource is not interpreted. For example within a <script> object context, JavaScript MIME types are expected (like *application/javascript*) in the Content-Type header.

Sensitive Code Example

In Express.js application the code is sensitive if, when using helmet, the `noSniff` middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet({
    noSniff: false, // Sensitive
  })
);
```

Compliant Solution

When using `helmet` in an Express.js application, the `noSniff` middleware should be enabled (it is also done by default):

```
const express = require('express');
const helmet= require('helmet');

let app = express();

app.use(helmet.noSniff());
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - X-Content-Type-Options
- blog.mozilla.org - Mitigating MIME Confusion Attacks in Firefox

how_to_fix

Recommended Secure Coding Practices

Implement X-Content-Type-Options header with *nosniff* value (the only existing value for this header) which is supported by all modern browsers and will prevent browsers from performing MIME type sniffing, so that in case of Content-Type header mismatch, the resource is not interpreted. For example within a <script> object context, JavaScript MIME types are expected (like *application/javascript*) in the Content-Type header.

Compliant Solution

When using `helmet` in an Express.js application, the `noSniff` middleware should be enabled (it is also done by default):

```
const express = require('express');
const helmet= require('helmet');

let app = express();

app.use(helmet.noSniff());
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - X-Content-Type-Options
- blog.mozilla.org - Mitigating MIME Confusion Attacks in Firefox

root_cause

MIME confusion attacks occur when an attacker successfully tricks a web-browser to interpret a resource as a different type than the one expected. To correctly interpret a resource (script, image, stylesheet …) web browsers look for the Content-Type header defined in the HTTP response received from the server, but often this header is not set or is set with an incorrect value. To avoid content-type mismatch and to provide the best user experience, web browsers try to deduce the right content-type, generally by inspecting the content of the resources (the first bytes). This "guess mechanism" is called MIME type sniffing.

Attackers can take advantage of this feature when a website ("example.com" here) allows to upload arbitrary files. In that case, an attacker can upload a malicious image *fakeimage.png* (containing malicious JavaScript code or a polyglot content file) such as:

```
<script>alert(document.cookie)</script>
```

When the victim will visit the website showing the uploaded image, the malicious script embedded into the image will be executed by web browsers performing MIME type sniffing.

Disabling strict HTTP no-referrer policy is security-sensitive

Clave: javascript:S5736

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- Confidential information exists in URLs.
- Semantic of HTTP methods is not respected (eg: use of a GET method instead of POST when the state of the application is changed).

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet `referrerPolicy` middleware is disabled or used with `no-referrer-when-downgrade` or `unsafe-url`:

```
const express = require('express');
const helmet = require('helmet');

app.use(
  helmet.referrerPolicy({
    policy: 'no-referrer-when-downgrade' // Sensitive: no-referrer-when-downgrade is used
  })
);
```

how_to_fix

Recommended Secure Coding Practices

Confidential information should not be set inside URLs (GET requests) of the application and a safe (ie: different from `unsafe-url` or `no-referrer-when-downgrade`) referrer-Policy header, to control how much information is included in the referer header, should be used.

Compliant Solution

In Express.js application a secure solution is to user the helmet referrer policy middleware set to `no-referrer`:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.referrerPolicy({
    policy: 'no-referrer' // Compliant
  })
);
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Referrer-Policy
- developer.mozilla.org - Referer header: privacy and security concerns
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

root_cause

HTTP header referer contains a URL set by web browsers and used by applications to track from where the user came from, it’s for instance a relevant value for web analytic services, but it can cause serious privacy and security problems if the URL contains confidential information. Note that Firefox for instance, to prevent data leaks, removes path information in the Referer header while browsing privately.

Suppose an e-commerce website asks the user his credit card number to purchase a product:

```
<html>
<body>
<form action="/valid_order" method="GET">
Type your credit card number to purchase products:
<input type=text id="cc" value="1111-2222-3333-4444">
<input type=submit>
</form>
</body>
```

When submitting the above HTML form, a HTTP GET request will be performed, the URL requested will be https://example.com/valid_order?cc=1111-2222-3333-4444 with credit card number inside and it’s obviously not secure for these reasons:

- URLs are stored in the history of browsers.
- URLs could be accidentally shared when doing copy/paste actions.
- URLs can be stolen if a malicious person looks at the computer screen of an user.

In addition to these threats, when further requests will be performed from the "valid_order" page with a simple legitimate embedded script like that:

```
<script src="https://webanalyticservices_example.com/track">
```

The referer header which contains confidential information will be send to a third party web analytic service and cause privacy issue:

```
GET /track HTTP/2.0
Host: webanalyticservices_example.com
Referer: https://example.com/valid_order?cc=1111-2222-3333-4444
```

default

HTTP header referer contains a URL set by web browsers and used by applications to track from where the user came from, it's for instance a relevant value for web analytic services, but it can cause serious privacy and security problems if the URL contains confidential information. Note that Firefox for instance, to prevent data leaks, removes path information in the Referer header while browsing privately.

Suppose an e-commerce website asks the user his credit card number to purchase a product:

```
<html>
<body>
<form action="/valid_order" method="GET">
Type your credit card number to purchase products:
<input type=text id="cc" value="1111-2222-3333-4444">
<input type=submit>
</form>
</body>
```

When submitting the above HTML form, a HTTP GET request will be performed, the URL requested will be https://example.com/valid_order?cc=1111-2222-3333-4444 with credit card number inside and it's obviously not secure for these reasons:

- URLs are stored in the history of browsers.
- URLs could be accidentally shared when doing copy/paste actions.
- URLs can be stolen if a malicious person looks at the computer screen of an user.

In addition to these threats, when further requests will be performed from the "valid_order" page with a simple legitimate embedded script like that:

```
<script src="https://webanalyticservices_example.com/track">
```

The referer header which contains confidential information will be send to a third party web analytic service and cause privacy issue:

```
GET /track HTTP/2.0
Host: webanalyticservices_example.com
Referer: https://example.com/valid_order?cc=1111-2222-3333-4444
```

Ask Yourself Whether

- Confidential information exists in URLs.
- Semantic of HTTP methods is not respected (eg: use of a GET method instead of POST when the state of the application is changed).

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Confidential information should not be set inside URLs (GET requests) of the application and a safe (ie: different from `unsafe-url` or `no-referrer-when-downgrade`) referrer-Policy header, to control how much information is included in the referer header, should be used.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet `referrerPolicy` middleware is disabled or used with `no-referrer-when-downgrade` or `unsafe-url`:

```
const express = require('express');
const helmet = require('helmet');

app.use(
  helmet.referrerPolicy({
    policy: 'no-referrer-when-downgrade' // Sensitive: no-referrer-when-downgrade is used
  })
);
```

Compliant Solution

In Express.js application a secure solution is to user the helmet referrer policy middleware set to `no-referrer`:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.referrerPolicy({
    policy: 'no-referrer' // Compliant
  })
);
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Referrer-Policy
- developer.mozilla.org - Referer header: privacy and security concerns
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

Disabling Strict-Transport-Security policy is security-sensitive

Clave: javascript:S5739

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The website is accessible with the unencrypted HTTP protocol.

There is a risk if you answered yes to this question.

Sensitive Code Example

In Express.js application the code is sensitive if the helmet or hsts middleware are disabled or used without recommended values:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.hsts({
  maxAge: 3153600, // Sensitive, recommended >= 15552000
  includeSubDomains: false // Sensitive, recommended 'true'
}));
```

default

When implementing the HTTPS protocol, the website mostly continue to support the HTTP protocol to redirect users to HTTPS when they request a HTTP version of the website. These redirects are not encrypted and are therefore vulnerable to man in the middle attacks. The Strict-Transport-Security policy header (HSTS) set by an application instructs the web browser to convert any HTTP request to HTTPS.

Web browsers that see the Strict-Transport-Security policy header for the first time record information specified in the header:

- the `max-age` directive which specify how long the policy should be kept on the web browser.
- the `includeSubDomains` optional directive which specify if the policy should apply on all sub-domains or not.
- the `preload` optional directive which is not part of the HSTS specification but supported on all modern web browsers.

With the `preload` directive the web browser never connects in HTTP to the website and to use this directive, it is required to submit the concerned application to a preload service maintained by Google.

Ask Yourself Whether

- The website is accessible with the unencrypted HTTP protocol.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement Strict-Transport-Security policy header, it is recommended to apply this policy to all subdomains (`includeSubDomains`) and for at least 6 months (`max-age=15552000`) or even better for 1 year (`max-age=31536000`).

Sensitive Code Example

In Express.js application the code is sensitive if the helmet or hsts middleware are disabled or used without recommended values:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.hsts({
  maxAge: 3153600, // Sensitive, recommended >= 15552000
  includeSubDomains: false // Sensitive, recommended 'true'
}));
```

Compliant Solution

In Express.js application a standard way to implement HSTS is with the helmet or hsts middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.hsts({
  maxAge: 31536000,
  includeSubDomains: true
})); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Strict Transport Security

root_cause

When implementing the HTTPS protocol, the website mostly continue to support the HTTP protocol to redirect users to HTTPS when they request a HTTP version of the website. These redirects are not encrypted and are therefore vulnerable to man in the middle attacks. The Strict-Transport-Security policy header (HSTS) set by an application instructs the web browser to convert any HTTP request to HTTPS.

Web browsers that see the Strict-Transport-Security policy header for the first time record information specified in the header:

- the `max-age` directive which specify how long the policy should be kept on the web browser.
- the `includeSubDomains` optional directive which specify if the policy should apply on all sub-domains or not.
- the `preload` optional directive which is not part of the HSTS specification but supported on all modern web browsers.

With the `preload` directive the web browser never connects in HTTP to the website and to use this directive, it is required to submit the concerned application to a preload service maintained by Google.

how_to_fix

Recommended Secure Coding Practices

Implement Strict-Transport-Security policy header, it is recommended to apply this policy to all subdomains (`includeSubDomains`) and for at least 6 months (`max-age=15552000`) or even better for 1 year (`max-age=31536000`).

Compliant Solution

In Express.js application a standard way to implement HSTS is with the helmet or hsts middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.hsts({
  maxAge: 31536000,
  includeSubDomains: true
})); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Strict Transport Security

Allowing browsers to perform DNS prefetching is security-sensitive

Clave: javascript:S5743

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule is deprecated, and will eventually be removed.

By default, web browsers perform DNS prefetching to reduce latency due to DNS resolutions required when an user clicks links from a website page.

For instance on example.com the hyperlink below contains a cross-origin domain name that must be resolved to an IP address by the web browser:

```
<a href="https://otherexample.com">go on our partner website</a>
```

It can add significant latency during requests, especially if the page contains many links to cross-origin domains. DNS prefetch allows web browsers to perform DNS resolving in the background before the user clicks a link. This feature can cause privacy issues because DNS resolving from the user’s computer is performed without his consent if he doesn’t intent to go to the linked website.

On a complex private webpage, a combination "of unique links/DNS resolutions" can indicate, to a eavesdropper for instance, that the user is visiting the private page.

assess_the_problem

Ask Yourself Whether

- Links to cross-origin domains could result in leakage of confidential information about the user’s navigation/behavior of the website.

There is a risk if you answered yes to this question.

Sensitive Code Example

In Express.js application the code is sensitive if the dns-prefetch-control middleware is disabled or used without the recommended value:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.dnsPrefetchControl({
    allow: true // Sensitive: allowing DNS prefetching is security-sensitive
  })
);
```

default

This rule is deprecated, and will eventually be removed.

By default, web browsers perform DNS prefetching to reduce latency due to DNS resolutions required when an user clicks links from a website page.

For instance on example.com the hyperlink below contains a cross-origin domain name that must be resolved to an IP address by the web browser:

```
<a href="https://otherexample.com">go on our partner website</a>
```

It can add significant latency during requests, especially if the page contains many links to cross-origin domains. DNS prefetch allows web browsers to perform DNS resolving in the background before the user clicks a link. This feature can cause privacy issues because DNS resolving from the user’s computer is performed without his consent if he doesn’t intent to go to the linked website.

On a complex private webpage, a combination "of unique links/DNS resolutions" can indicate, to a eavesdropper for instance, that the user is visiting the private page.

Ask Yourself Whether

- Links to cross-origin domains could result in leakage of confidential information about the user’s navigation/behavior of the website.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement X-DNS-Prefetch-Control header with an *off* value but this could significantly degrade website performances.

Sensitive Code Example

In Express.js application the code is sensitive if the dns-prefetch-control middleware is disabled or used without the recommended value:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.dnsPrefetchControl({
    allow: true // Sensitive: allowing DNS prefetching is security-sensitive
  })
);
```

Compliant Solution

In Express.js application the dns-prefetch-control or helmet middleware is the standard way to implement `X-DNS-Prefetch-Control` header:

```
const express = require('express');
const helmet = require('helmet');
```

```
let app = express();

app.use(
  helmet.dnsPrefetchControl({
    allow: false // Compliant
  })
);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - X-DNS-Prefetch-Control
- developer.mozilla.org - Using dns-prefetch

how_to_fix

Recommended Secure Coding Practices

Implement X-DNS-Prefetch-Control header with an *off* value but this could significantly degrade website performances.

Compliant Solution

In Express.js application the dns-prefetch-control or helmet middleware is the standard way to implement `X-DNS-Prefetch-Control` header:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
  helmet.dnsPrefetchControl({
    allow: false // Compliant
  })
);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - X-DNS-Prefetch-Control
- developer.mozilla.org - Using dns-prefetch

Alternatives in regular expressions should be grouped when used with anchors

Clave: javascript:S5850

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Regular expressions
- MDN web docs - Assertions
- MDN web docs - Disjunction: `|`
- MDN web docs - Non-capturing group: `(?:...)`

root_cause

Regular expressions are used for pattern matching within strings. They can be defined upon special characters, meaning symbols or metacharacters with a reserved meaning that convey specific instructions to the regex engine. These characters are not treated as literals but have special functions in defining patterns, among which stand out anchors and disjunctions.

- An anchor allows you to match positions in the input string rather than matching specific characters. Anchors help you identify specific locations within the string where a pattern should start (`^`) or end (`$`).
- A disjunction, also known as alternatives, represented by the vertical bar (`|`) allows you to specify multiple alternative patterns that the regex engine will attempt to match in the input string.

Mixing anchors with alternatives in regular expressions can lead to confusion due to their precedence rules. Alternatives (`|`) have a lower precedence than anchors (`^` and `$`). As a result, if you don't use non-capturing groups `(?:...)` to group the alternatives properly, the anchors might apply to the ends only rather than the entire disjunction, which could not be the initial intent.

```
const regex = /^a|b|c$/; // Noncompliant: '^' applies to 'a' and '$' applies to 'c'
```

You should group the disjunction with parentheses denoting non-capturing groups so that the anchors apply to all alternatives.

```
const regex = /^(?:a|b|c)$/;
```

Alternatively, you can distribute the anchors to each alternative of the disjunction.

```
const regex = /^a$|^b$|^c$/;
```

If the precedence of the operators is understood and the intention is to apply the anchors to only the ends, use parentheses to make it explicit.

```
const regex = /(?:^a)|b|(?:c$)/;
```

Using slow regular expressions is security-sensitive

Clave: javascript:S5852

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The input is user-controlled.
- The input size is not restricted to a small number of characters.
- There is no timeout in place to limit the regex evaluation time.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

The regex evaluation will never end:

```
/(a+)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Sensitive
```

root_cause

Most of the regular expression engines use backtracking to try all possible execution paths of the regular expression when evaluating an input, in some cases it can cause performance issues, called **catastrophic backtracking** situations. In the worst case, the complexity of the regular expression is exponential in the size of the input, this means that a small carefully-crafted input (like 20 chars) can trigger catastrophic backtracking and cause a denial of service of the application. Super-linear regex complexity can lead to the same impact too with, in this case, a large carefully-crafted input (thousands chars).

This rule determines the runtime complexity of a regular expression and informs you if it is not linear.

how_to_fix

Recommended Secure Coding Practices

To avoid catastrophic backtracking situations, make sure that none of the following conditions apply to your regular expression.

In all of the following cases, catastrophic backtracking can only happen if the problematic part of the regex is followed by a pattern that can fail, causing the backtracking to actually happen.

- If you have a repetition `r*` or `r*?`, such that the regex `r` could produce different possible matches (of possibly different lengths) on the same input, the worst case matching time can be exponential. This can be the case if `r` contains optional parts, alternations or additional repetitions (but not if the repetition is written in such a way that there's only one way to match it).
- If you have multiple repetitions that can match the same contents and are consecutive or are only separated by an optional separator or a separator that can be matched by both of the repetitions, the worst case matching time can be polynomial (O(n^c) where c is the number of problematic repetitions). For example `a*b*` is not a problem because `a*` and `b*` match different things and `a*_a*` is not a problem because the repetitions are separated by a `'_'` and can't match that `'_'`. However, `a*a*` and `.*_.*` have quadratic runtime.
- If the regex is not anchored to the beginning of the string, quadratic runtime is especially hard to avoid because whenever a match fails, the regex engine will try again starting at the next index. This means that any unbounded repetition, if it's followed by a pattern that can fail, can cause quadratic runtime on some inputs. For example `str.split(/\s*,/)` will run in quadratic time on strings that consist entirely of spaces (or at least contain large sequences of spaces, not followed by a comma).

In order to rewrite your regular expression without these patterns, consider the following strategies:

- If applicable, define a maximum number of expected repetitions using the bounded quantifiers, like `{1,5}` instead of `+` for instance.
- Refactor nested quantifiers to limit the number of way the inner group can be matched by the outer quantifier, for instance this nested quantifier situation `(ba+)+` doesn't cause performance issues, indeed, the inner group can be matched only if there exists exactly one `b` char per repetition of the group.
- Optimize regular expressions by emulating *possessive quantifiers* and *atomic grouping*.
- Use negated character classes instead of `.` to exclude separators where applicable. For example the quadratic regex `.*_.*` can be made linear by changing it to `[^_]*_.*`

Sometimes it's not possible to rewrite the regex to be linear while still matching what you want it to match. Especially when the regex is not anchored to the beginning of the string, for which it is quite hard to avoid quadratic runtimes. In those cases consider the following approaches:

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's RE2 or node-re2.
- Use multiple passes. This could mean pre- and/or post-processing the string manually before/after applying the regular expression to it or using multiple regular expressions. One example of this would be to replace `str.split(/\s*,\s*/)` with `str.split(",")` and then trimming the spaces from the strings as a second step.
- It is often possible to make the regex infallible by making all the parts that could fail optional, which will prevent backtracking. Of course this means that you'll accept more strings than intended, but this can be handled by using capturing groups to check whether the optional parts were matched or not and then ignoring the match if they weren't. For example the regex `x*y` could be replaced with `x*(y)?` and then the call to `str.match(regex)` could be replaced with `matched = str.match(regex)` and `matched[1] !== undefined`.

Compliant Solution

Possessive quantifiers do not keep backtracking positions, thus can be used, if possible, to avoid performance issues. Unfortunately, they are not supported in JavaScript, but one can still mimick them using lookahead assertions and backreferences:

```
/((?=(a+))\2)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Compliant
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-400 - Uncontrolled Resource Consumption
- CWE - CWE-1333 - Inefficient Regular Expression Complexity
- owasp.org - OWASP Regular expression Denial of Service - ReDoS
- stackstatus.net(archived) - Outage Postmortem - July 20, 2016
- regular-expressions.info - Runaway Regular Expressions: Catastrophic Backtracking
- docs.microsoft.com - Backtracking with Nested Optional Quantifiers

default

Most of the regular expression engines use backtracking to try all possible execution paths of the regular expression when evaluating an input, in some cases it can cause performance issues, called **catastrophic backtracking** situations. In the worst case, the complexity of the regular expression is exponential in the size of the input, this means that a small carefully-crafted input (like 20 chars) can trigger catastrophic backtracking and cause a denial of service of the application. Super-linear regex complexity can lead to the same impact too with, in this case, a large carefully-crafted input (thousands chars).

This rule determines the runtime complexity of a regular expression and informs you if it is not linear.

Ask Yourself Whether

- The input is user-controlled.
- The input size is not restricted to a small number of characters.
- There is no timeout in place to limit the regex evaluation time.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

To avoid catastrophic backtracking situations, make sure that none of the following conditions apply to your regular expression.

In all of the following cases, catastrophic backtracking can only happen if the problematic part of the regex is followed by a pattern that can fail, causing the backtracking to actually happen.

- If you have a repetition `r*` or `r*?`, such that the regex `r` could produce different possible matches (of possibly different lengths) on the same input, the worst case matching time can be exponential. This can be the case if `r` contains optional parts, alternations or additional repetitions (but not if the repetition is written in such a way that there's only one way to match it).
- If you have multiple repetitions that can match the same contents and are consecutive or are only separated by an optional separator or a separator that can be matched by both of the repetitions, the worst case matching time can be polynomial (O(n^c) where c is the number of problematic repetitions). For example `a*b*` is not a problem because `a*` and `b*` match different things and `a*_a*` is not a problem because the repetitions are separated by a `'_'` and can't match that `'_'`. However, `a*a*` and `.*_.*` have quadratic runtime.

- If the regex is not anchored to the beginning of the string, quadratic runtime is especially hard to avoid because whenever a match fails, the regex engine will try again starting at the next index. This means that any unbounded repetition, if it's followed by a pattern that can fail, can cause quadratic runtime on some inputs. For example `str.split(/\s*,/)` will run in quadratic time on strings that consist entirely of spaces (or at least contain large sequences of spaces, not followed by a comma).

In order to rewrite your regular expression without these patterns, consider the following strategies:

- If applicable, define a maximum number of expected repetitions using the bounded quantifiers, like `{1,5}` instead of `+` for instance.
- Refactor nested quantifiers to limit the number of way the inner group can be matched by the outer quantifier, for instance this nested quantifier situation `(ba+)+` doesn't cause performance issues, indeed, the inner group can be matched only if there exists exactly one `b` char per repetition of the group.
- Optimize regular expressions by emulating *possessive quantifiers* and *atomic grouping*.
- Use negated character classes instead of `.` to exclude separators where applicable. For example the quadratic regex `.*_.*` can be made linear by changing it to `[^_]*_.*`

Sometimes it's not possible to rewrite the regex to be linear while still matching what you want it to match. Especially when the regex is not anchored to the beginning of the string, for which it is quite hard to avoid quadratic runtimes. In those cases consider the following approaches:

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's RE2 or node-re2.
- Use multiple passes. This could mean pre- and/or post-processing the string manually before/after applying the regular expression to it or using multiple regular expressions. One example of this would be to replace `str.split(/\s*,\s*/)` with `str.split(",")` and then trimming the spaces from the strings as a second step.
- It is often possible to make the regex infallible by making all the parts that could fail optional, which will prevent backtracking. Of course this means that you'll accept more strings than intended, but this can be handled by using capturing groups to check whether the optional parts were matched or not and then ignoring the match if they weren't. For example the regex `x*y` could be replaced with `x*(y)?` and then the call to `str.match(regex)` could be replaced with `matched = str.match(regex)` and `matched[1] !== undefined`.

Sensitive Code Example

The regex evaluation will never end:

```
/(a+)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Sensitive
```

Compliant Solution

Possessive quantifiers do not keep backtracking positions, thus can be used, if possible, to avoid performance issues. Unfortunately, they are not supported in JavaScript, but one can still mimick them using lookahead assertions and backreferences:

```
/((?=(a+))\2)+$/.test(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!"
); // Compliant
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-400 - Uncontrolled Resource Consumption
- CWE - CWE-1333 - Inefficient Regular Expression Complexity
- owasp.org - OWASP Regular expression Denial of Service - ReDoS
- stackstatus.net(archived) - Outage Postmortem - July 20, 2016
- regular-expressions.info - Runaway Regular Expressions: Catastrophic Backtracking
- docs.microsoft.com - Backtracking with Nested Optional Quantifiers

---

Regular expressions should be syntactically valid

Clave: javascript:S5856

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Regular expressions have their own syntax that is understood by regular expression engines. Those engines will throw an exception at runtime if they are given a regular expression that does not conform to that syntax.

To avoid syntax errors, special characters should be escaped with backslashes when they are intended to be matched literally and references to capturing groups should use the correctly spelled name or number of the group.

To match a literal string, rather than a regular expression, either all special characters should be escaped or methods that don’t use regular expressions should be used.

Noncompliant code example

```
new RegExp("([");
str.match("([");
```

Compliant solution

```
new RegExp("\\(\\[");
str.match("\\(\\[");
str.replace("([", "{");
```

Tests should be stable

Clave: javascript:S5973

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Make your test stable so that it passes on the first try, or remove it.

Noncompliant code example

```
jest.retryTimes(3); // Noncompliant

describe('API.foo()', function() {
  it('should return 5 when computing ...', function() {
    doSomethingUnstable();
  });
});
```

resources

Documentation

- Jest docs - jest.retryTimes()
- Mocha docs - Retry tests

Articles & blog posts

- Spotify Engineering - Test Flakiness - Methods for identifying and dealing with flaky tests

root_cause

Unstable / flaky tests are tests which sometimes pass and sometimes fail, without any code change. Obviously, they slow down developments when developers have to rerun failed tests. However, the real problem is that you can’t completely trust these tests, they might fail for many different reasons and you don’t know if any of them will happen in production.

Some tools, such as Jest, enable developers to automatically retry flaky tests. This might be acceptable as a temporary solution, but it should eventually be fixed. The more flaky tests you add, the more chances there are for a bug to arrive in production.

This rule raises an issue when these functions are called with a value higher than `0`: * `jest.retry()` * `this.retries()` inside a Mocha test case

---

DOM elements with ARIA roles should have a valid non-abstract role

Clave: javascript:S6821

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

This rule checks that when using the `role` property in DOM elements, its value is a valid non-abstract ARIA role.

This rule does not cover non-DOM elements, such as custom components.

how_to_fix

Check that each element with a defined ARIA role has a valid non-abstract value.

```
<div role="meth" aria-label="a^{2} + b^{2} = c^{2}">
  a<sup>2</sup> + b<sup>2</sup> = c<sup>2</sup>
</div>
```

To fix the code use a valid value for the ARIA role attribute.

```
<div role="math" aria-label="a^{2} + b^{2} = c^{2}">
  a<sup>2</sup> + b<sup>2</sup> = c<sup>2</sup>
</div>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

No redundant ARIA role

Clave: javascript:S6822

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

In HTML, certain elements have default roles. Default roles, also known as implicit roles, are roles that are inherently associated with certain HTML elements. These roles provide information about what an element does or the type of content it contains, which is especially useful for assistive technologies like screen readers.

For example, a `<button>` element has a default role of `button`. If you explicitly define the role of a `<button>` element as `button`, it’s considered redundant because it’s the default role of that element.

```
<button role="button" onClick={handleClick}>OK</button>
```

Remove ARIA role attributes when they are redundant.

```
<button onClick={handleClick}>OK</button>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

DOM elements with the `aria-activedescendant` property should be accessible via the tab key

Clave: javascript:S6823

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

The `aria-activedescendant` attribute is used to enhance the accessibility of composite widgets by managing focus within them. It allows a parent element to retain active document focus while indicating which of its child elements has secondary focus. This attribute is particularly useful in interactive components like search typeahead select lists, where the user can navigate through a list of options while continuing to type in the input field.

This rule checks that DOM elements with the `aria-activedescendant` property either have an inherent tabIndex or declare one.

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)
- MDN web docs - `aria-activedescendant` attribute
- MDN web docs - `tabIndex` attribute

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2
- W3C - Composite role

how_to_fix

Make sure that DOM elements with the `aria-activedescendant` property have a `tabIndex` property, or use an element with an inherent one.

Noncompliant code example

```
<div aria-activedescendant={descendantId}>
  {content}
</div>
```

Compliant solution

```
<div aria-activedescendant={descendantId} tabIndex={0}>
  {content}
</div>
```

No ARIA role or property for unsupported DOM elements

Clave: javascript:S6824

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Check if you are using ARIA roles or `aria-*` attributes in unsupported DOM elements.

```
<title aria-hidden="false">My beautiful web page</title>
```

To fix the code, remove the extra ARIA role or `aria-*` attributes from the unsupported DOM elements.

```
<title>My beautiful web page</title>
```

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

This rule checks that ARIA roles or `aria-*` attributes are not used in unsupported DOM elements, which are mostly invisible such as `meta`, `html` or `head`.

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

Focusable elements should not have "aria-hidden" attribute

Clave: javascript:S6825

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Check if the element is focusable. Focusable elements should not have `aria-hidden` attribute.

Noncompliant code example

```
<button aria-hidden="true">Click me</button>
```

Remove `aria-hidden` attribute.

Compliant solution

```
<button>Click me</button>
```

root_cause

ARIA (Accessible Rich Internet Applications) is a set of attributes that define ways to make web content and web applications more accessible to people with disabilities. The `aria-hidden` attribute is used to indicate that an element and all of its descendants are not visible or perceivable to any user as implemented by assistive technologies.

However, when `aria-hidden` is used on a focusable element, it can create a confusing and inaccessible experience for screen reader users. This is because the element will still be included in the tab order, so a screen reader user can navigate to it, but it will not be announced by the screen reader due to the `aria-hidden` attribute.

This rule ensures that focusable elements are not hidden from screen readers using the `aria-hidden` attribute.

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - aria-hidden attribute (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

Anchors should contain accessible content

Clave: javascript:S6827

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `<a>`: The Anchor element
- MDN web docs - `aria-hidden` attribute

Standards

- W3C - Link purpose

root_cause

Anchors, represented by the a tag in HTML, usually contain a hyperlink that users can click to navigate to different sections of a website or different websites altogether.

However, when anchors do not have content or when the content is hidden from screen readers using the `aria-hidden` property, it creates a significant accessibility issue. If an anchor’s content is hidden or non-existent, visually impaired users may not be able to understand the purpose of the anchor or navigate the website effectively.

This rule checks that anchors do not use the `aria-hidden` property and have content provided either between the tags or as `aria-label` or `title` property.

how_to_fix

Ensure that anchors either have content or an `aria-label` or `title` attribute, and they should not use the `aria-hidden` property.

Noncompliant code example

```
<a aria-hidden>link to my site</a>
```

Compliant solution

```
<a>link to my site</a>
```

Mergeable "if" statements should be combined

Clave: javascript:S1066

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Nested code - blocks of code inside blocks of code - is eventually necessary, but increases complexity. This is why keeping the code as flat as possible, by avoiding unnecessary nesting, is considered a good practice.

Merging `if` statements when possible will decrease the nesting of the code and improve its readability.

Code like

```
if (x != undefined) {
  if (y === 2) {
    // ...
  }
}
```

Will be more readable as

```
if (x != undefined && y === 2) {
  // ...
}
```

how_to_fix

If merging the conditions seems to result in a more complex code, extracting the condition or part of it in a named function or variable is a better approach to fix readability.

Noncompliant code example

```
if (file != undefined) {
  if (file.isFile() || file.isDirectory()) {        // Noncompliant
    /* ... */
  }
}
```

Compliant solution

```
function isFileOrDirectory(File file) {
  return file.isFile() || file.isDirectory();
}

/* ... */
```

```
if (file. != undefined && isFileOrDirectory(file)) { // Compliant
  /* ... */
}
```

Expressions should not be too complex

Clave: javascript:S1067

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

The complexity of an expression is defined by the number of &&, `||` and `condition ? ifTrue : ifFalse` operators it contains.

A single expression’s complexity should not become too high to keep the code readable.

Unused private class members should be removed

Clave: javascript:S1068

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The private class members were introduced in ES2022 and use # (hash) symbol prefix. It is possible to declare private fields, methods, getters and setters as well as their static counterparts. The private members are only accessible from within the current class body and aren’t inherited by subclasses. They also cannot be removed with `delete` operator.

```
class MyClass {
    #foo = 123;
    bar(){
        return this.#foo; // ok
    }
}

const obj = new MyClass();
obj.#foo = 345; // error: #foo is not accessible outside of the class
```

Private class members that are declared and not used anywhere in the code are most likely an error due to incomplete refactoring and should be corrected or removed.

```
class MyClass {
    #privateField1;
    #privateField2;  // Noncompliant: #privateField2 is unused
    #privateMethod(){} // Noncompliant: #privateMethod is unused
    publicMethod(){
        return this.#privateField1;
    }
}
```

To fix the code remove unused private member of the class.

```
class MyClass {
    #privateField1;
    publicMethod(){
        return this.#privateField1;
    }
}
```

resources

Documentation

- MDN web docs - Private class features
- MDN web docs - Classes

Redundant pairs of parentheses should be removed

Clave: javascript:S1110

Severidad: MAJOR

Estado: ⚠ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

Parentheses can disambiguate the order of operations in complex expressions and make the code easier to understand.

```
a = (b * c) + (d * e); // Compliant: the intent is clear.
```

Redundant parentheses are parenthesis that do not change the behavior of the code, and do not clarify the intent. They can mislead and complexify the code. They should be removed.

Noncompliant code example

```
let x = ((y / 2 + 1));  // Noncompliant

if (a && ((x + y > 0))) {  // Noncompliant
  return ((x + 1));  // Noncompliant
}
```

Compliant solution

```
let x = (y / 2 + 1);

if (a && (x + y > 0)) {
  return (x + 1);
}
```

Extra semicolons should be removed

Clave: javascript:S1116

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

Extra semicolons (`;`) are usually introduced by mistake, for example because:

- It was meant to be replaced by an actual statement, but this was forgotten.
- There was a typo which lead the semicolon to be doubled, i.e. `;;`.
- There was a misunderstanding about where semicolons are required or useful.

Noncompliant code example

```
var x = 1;; // Noncompliant

function foo() {
};  // Noncompliant
```

Compliant solution

```
var x = 1;

function foo() {
}
```

Exceptions

This rule does not apply when the semicolon is after a line break and before `(` or `[` as it is often used in semicolon-less style.

```
var hello = 'Hello'
var world = 'World!'
var helloWorld = hello + ' ' + world
;[...helloWorld].forEach(c => console.log(c))

var a = 1
var b = 2 * a
;(a + b).toString()
```

introduction

This rule is deprecated, and will eventually be removed.

Variables should not be shadowed

Clave: javascript:S1117

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Variable shadowing happens when a variable declared in a specific scope has the same name as a variable in an outer scope.

This can lead to three main problems:

- Confusion: The same name can refer to different variables in different parts of the scope, making the code hard to read and understand.
- Unintended Behavior: You might accidentally use the wrong variable, leading to hard-to-detect bugs.
- Maintenance Issues: If the inner variable is removed or renamed, the code’s behavior might change unexpectedly because the outer variable is now being used.

To avoid these problems, rename the shadowing, shadowed, or both variables to accurately represent their purpose with unique and meaningful names.

Note that functions in JavaScript are first-class citizens. This means that they possess the same attributes as variables, including the ability to shadow other variables and, conversely, be shadowed by them.

Noncompliant code example

The example below shows the typical situations in which shadowing can occur.

```
function outer(items) {
  var counter = 0;

  function inner(items) { // Noncompliant: the parameter "items" is shadowed.
    var counter = counter + 1; // Noncompliant: the outer "counter" is shadowed.
  }

  inner(items);

  return counter; // returns 0
}

function search(items, match) { // Noncompliant: the function "match" (below) is shadowed.
  // ...
}

function match(value, key) {
  // ...
}
```

resources

Related rules

- S2814 - Variables and functions should not be redeclared

Labels should not be used

Clave: javascript:S1119

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, labels are identifiers that allow you to name blocks of code, such as loops and conditional statements. They are used in conjunction with statements like `break` and `continue` to control the flow of execution within nested loops and conditionals.

It’s worth noting that labels are not widely used in modern JavaScript programming because they can lead to complex and hard-to-maintain code. In most cases, there are better alternatives to achieve the desired control flow without resorting to labels.

```
myLabel: {
  let x = doSomething();
  if (x > 0) {
    break myLabel;
  }
  doSomethingElse();
}
```

If you find yourself using labels, you should reevaluate your code structure and explore other options for better code clarity and maintainability.

```
let x = doSomething();
if (x <= 0) {
  doSomethingElse();
}
```

resources

Documentation

- MDN web docs - label
- MDN web docs - `break`
- MDN web docs - `continue`

Assignments should not be made from within sub-expressions

Clave: javascript:S1121

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Making assignments within sub-expressions can hinder the clarity of source code.

This practice essentially gives a side-effect to a larger expression, thus making it less readable. This often leads to confusion and potential errors.

Extracting assignments into separate statements is encouraged to keep the code clear and straightforward.

Noncompliant code example

```
if (val = value() && check()) { // Noncompliant
  // ...
}
```

Compliant solution

```
val = value();
if (val && check()) {
  // ...
}
```

root_cause

A common code smell that can hinder the clarity of source code is making assignments within sub-expressions. This practice involves assigning a value to a variable inside a larger expression, such as within a loop or a conditional statement.

This practice essentially gives a side-effect to a larger expression, thus making it less readable. This often leads to confusion and potential errors.

Moreover, using chained assignments in declarations is also dangerous because one may accidentally create global variables. Consider the following code snippet: `let x = y = 1;`. If y is not declared, it will be hoisted as global.

Exceptions

The rule does not raise issues for the following patterns:

- chained assignments: `a = b = c = 0;`
- relational assignments: `(a = 0) != b`
- sequential assignments: `a = 0, b = 1, c = 2`
- assignments in lambda body: `() => a = 0`
- conditional assignment idiom: `a || (a = 0)`
- assignments in (do-)while conditions: `while (a = 0);`

resources

- CWE - CWE-481 - Assigning instead of Comparing

Boolean literals should not be used in comparisons

Clave: javascript:S1125

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A boolean literal can be represented in two different ways: `true` or `false`. They can be combined with logical operators (`!, &&, ||, ==, !=`) to produce logical expressions that represent truth values. However, comparing a boolean literal to a variable or expression that evaluates to a boolean value is unnecessary and can make the code harder to read and understand. The more complex a boolean expression is, the harder it will be for developers to understand its meaning and expected behavior, and it will favour the introduction of new bugs.

how_to_fix

Remove redundant boolean literals from expressions to improve readability and make the code more maintainable.

```
if (someValue == true) { /* ... */ } // Noncompliant: Redundant comparison
if (someBooleanValue != true) { /* ... */ } // Noncompliant: Redundant comparison
if (booleanMethod() || false) { /* ... */ }  // Noncompliant: Redundant OR
doSomething(!false); // Noncompliant: Redundant negation
```

Remove redundant boolean literals to improve readability.

```
if (someValue) { /* ... */ }
if (!someBooleanValue) { /* ... */ }
if (booleanMethod()) { /* ... */ }
doSomething(true);
```

resources

Documentation

- MDN web docs - Equality comparisons and sameness
- MDN web docs - Boolean

Return of boolean expressions should not be wrapped into an "if-then-else" statement

Clave: javascript:S1126

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `if...else` statement is used to make decisions based on the truthiness of a boolean expression, and the `if` block executes when the expression is truthy, while the `else` block executes when the expression is falsy.

Wrapping a boolean expression in an `if...else` statement and returning `true` or `false` in the respective blocks is redundant and unnecessary. It can also make the code harder to maintain, as it adds unnecessary lines of code that need to be read and understood.

```
if (expression) {
  return true;
} else {
  return false;
}
```

Simplify the code and return the boolean expression (or its negation) directly to make the code more concise and easier to read and maintain.

```
return expression;
```

If the caller expects a boolean and the result of the expression is not a boolean, use double negation for proper conversion.

```
return !!expression;
```

resources

Documentation

- MDN web docs - Boolean
- MDN web docs - `if...else`
- MDN web docs - Double NOT (`!!`)

Unnecessary imports should be removed

Clave: javascript:S1128

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Unnecessary imports refer to importing modules, libraries, or dependencies that are not used or referenced anywhere in the code. These imports do not contribute to the functionality of the application and only add extra weight to the JavaScript bundle, leading to potential performance and maintainability issues.

```
import A from 'a'; // Noncompliant: The imported symbol 'A' isn't used
import { B1 } from 'b';

console.log(B1);
```

To mitigate the problems associated with unnecessary imports, you should regularly review and remove any imports that are not being used. Modern JavaScript build tools and bundlers often provide features like tree shaking, which eliminates unused code during the bundling process, resulting in a more optimized bundle size.

```
import { B1 } from 'b';

console.log(B1);
```

resources

Documentation

- MDN web docs - `import`

Related rules

- S1481 - Unused local variables and functions should be removed

---

Jump statements should not occur in "finally" blocks

Clave: javascript:S1143

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - The `finally` block
- MDN web docs - `try...catch`
- MDN web docs - `return`
- MDN web docs - `throw`
- MDN web docs - `break`
- MDN web docs - `continue`

root_cause

The `finally` block is a part of a `try…catch…finally` statement, which allows you to handle errors and perform cleanup operations regardless of whether an exception is thrown or not. The `finally` block is executed regardless of whether an exception occurs or not, and it is placed after the `try` and `catch` blocks.

Having a jump statement, such as `return`, `break`, `continue`, or `throw`, inside a `finally` block can lead to unexpected and undesirable behavior, making your code difficult to understand and maintain. While it's not inherently forbidden to use jump statements in `finally` blocks, it is generally discouraged for the following reasons:

- The primary purpose of the `finally` block is to ensure cleanup operations and code that must run regardless of the outcome, such as releasing resources or closing connections. If you use a `return` statement inside the finally block, it will override any previous `return` statements in the `try` or `catch` blocks. This can lead to unexpected values being returned from a function.
- Jump statements like `break`, `continue`, or even another `throw` inside the `finally` block can alter the normal control flow of the program. This can make it difficult to reason about the behavior of the code and may introduce subtle bugs that are hard to detect.

- If a `return` or `throw` statement inside the `finally` block causes a new exception or alters the return value, it can hide or suppress the original exception or return value from the `try` or `catch` blocks. This can make it challenging to identify the actual cause of an error.
- Code that uses jump statements in `finally` blocks can be hard to read and understand, especially for other developers who might not be familiar with the unusual control flow. Such code can lead to maintenance issues and make it harder to debug and maintain the application in the long run.

This rule reports on all usages of jump statements from a `finally` block. Even if it’s guaranteed that no unhandled exception can happen in `try` or `catch` blocks, it’s not recommended to use any jump statements inside the `finally` block to have the logic there limited to the "cleanup".

```
async function foo() {
    let result, connection;
    try {
        connection = await connect();
        result = connection.send(1);
    } catch(err) {
        console.error(err.message);
    } finally {
        if (connection) {
            connection.close();
        }
        return result; // Noncompliant: Jump statement 'return' in the 'finally' block
    }
}
```

While there might be rare cases where using jump statements in a `finally` block is necessary, it’s generally recommended to avoid it whenever possible. Instead, use the `finally` block only for cleanup operations and critical tasks that should always be executed, regardless of exceptions or return values.

```
async function foo() {
    let result, connection;
    try {
        connection = await connect();
        result = connection.send(1);
    } catch(err) {
        console.error(err.message);
    } finally {
        if (connection) {
            connection.close();
        }
    }
    return result;
}
```

Results of operations on strings should not be ignored

Clave: javascript:S1154

Severidad: MAJOR

Estado: ⚠️ Depreciada

Descripción: No disponible

root_cause

Doing an operation on a string without using the result of the operation is useless and is certainly due to a misunderstanding.

Noncompliant code example

```
var str = "..."
str.toUpperCase(); // Noncompliant
```

Compliant solution

```
var str = "..."
str = str.toUpperCase();
```

introduction

This rule is deprecated; use S2201 instead.

Unused function parameters should be removed

Clave: javascript:S1172

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Having unused function parameters in your code can lead to confusion and misunderstanding of a developer’s intention. They reduce code readability and introduce the potential for errors. To avoid these problems, developers should remove unused parameters from function declarations.

Noncompliant code example

```
function doSomething(a, b) { // "a" is unused
  return compute(b);
}
```

Compliant solution

```
function doSomething(b) {
  return compute(b);
}
```

or

```
function doSomething(_a, b) {
  return compute(b);
}
```

root_cause

A typical code smell known as unused function parameters refers to parameters declared in a function but not used anywhere within the function’s body. While this might seem harmless at first glance, it can lead to confusion and potential errors in your code. Disregarding the values passed to such parameters, the function’s behavior will be the same, but the programmer’s intention won’t be clearly expressed anymore. Therefore, removing function parameters that are not being utilized is considered best practice.

Exceptions

When `arguments` is used in the function body, no parameter is reported as unused.

```
function doSomething(a, b, c) {
  compute(arguments);
}
```

The rule also ignores all parameters with names starting with an underscore (_). This practice is often used to indicate that some parameter is intentionally unused. This practice is frequently seen in the TypeScript compiler, for example.

```
function doSomething(_a, b) {
  return compute(b);
}
```

Functions should not be empty

Clave: javascript:S1186

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

An empty method is generally considered bad practice and can lead to confusion, readability, and maintenance issues. Empty methods bring no functionality and are misleading to others as they might think the method implementation fulfills a specific and identified requirement.

There are several reasons for a method not to have a body:

- It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production.
- It is not yet, or never will be, supported. In this case an exception should be thrown.
- The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override.

Exceptions

This does not raise an issue in the following cases:

- Arrow functions as they can denote default values.
- Functions with a name starting with the prefix `on` like `onClick`.
- Functions whose name includes `noop`.
- Constructors as it is already covered by S6647.

```
static defaultProps = {
  listStyle: () => {}
};
```

```
function onClick() {
}

function myNoopFunction() {
}

class C {
  constructor() {}
}
```

resources

Documentation

- MDN web docs - Functions

Related rules

- S6647 - Unnecessary constructors should be removed

how_to_fix

Noncompliant code example

```
function shouldNotBeEmpty() {  // Noncompliant - method is empty
}

function notImplemented() {  // Noncompliant - method is empty
}

function emptyOnPurpose() {  // Noncompliant - method is empty
}
```

Compliant solution

```
function shouldNotBeEmpty() {
  doSomething();
}

function notImplemented() {
  throw new Error("notImplemented() cannot be performed because ...");
}

function emptyOnPurpose() {
  // comment explaining why the method is empty
}
```

A "while" loop should be used instead of a "for" loop

Clave: javascript:S1264

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `for`
- MDN web docs - `while`

root_cause

A `for` loop is a type of loop construct that allows a block of code to be executed repeatedly for a fixed number of times. The `for` loop is typically used when the number of iterations is known in advance, and consists of three parts:

- The initialization statement is executed once at the beginning of the loop, and is used to initialize the loop counter or any other variables that may be used in the loop.
- The loop condition is evaluated at the beginning of each iteration, and if it is `true`, the code inside the loop is executed.
- The update statement is executed at the end of each iteration, and is used to update the loop counter or any other variables that may be used in the loop.

```
for (initialization; condition; update) { /*...*/ }
```

All three statements are optional. However, when the initialization and update statements are not used, it can be unclear to the reader what the loop counter is and how it is being updated. This can make the code harder to understand and maintain.

```
for (;condition;) { /*...*/ } // Noncompliant: Only the condition is specified
```

When only the condition expression is defined in a `for` loop, a `while` loop should be used instead to increase readability. A `while` loop consists of a single loop condition and allows a block of code to be executed repeatedly as long as the specified condition is true.

```
while (condition) { /*...*/ }
```

Track uses of "NOSONAR" comments

Clave: javascript:S1291

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Any issue to quality rule can be deactivated with the `NOSONAR` marker. This marker is pretty useful to exclude false-positive results but it can also be used abusively to hide real quality flaws.

This rule raises an issue when `NOSONAR` is used.

Function call arguments should not start on new lines

Clave: javascript:S1472

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

JavaScript will automatically insert semicolons when parsing the code so invalid sequences can be "fixed" to valid syntax. This behavior, called "Automatic semicolon insertion" or **ASI**, makes semicolons at the end of statements optional and attempts to make JavaScript more approachable and convenient.

However, sometimes, relying on ASI can lead to unexpected results. ASI will only be triggered if a line break separates tokens that would otherwise produce invalid syntax. JavaScript will not insert semicolons if the next token can be parsed as part of a valid structure.

In the case of function call arguments, they are allowed to be on a separate line. But, depending on the developer’s intent and, especially when working with IIFE (or any other design pattern using Grouping operator), it can lead to errors and most likely *will* lead to questions for maintainers.

What was the initial intent of the developer?

1. Defining a function and then executing some unrelated code inside a closure?
2. Passing the second function as a parameter to the first one?

The first option will be the one chosen by the JavaScript interpreter.

```
const fn = function () {
  //...
}

(function () { // Noncompliant: function is passed as a parameter to fn
  //...
})();
```

By extension, and to improve readability, any kind of function call argument should not start on a new line.

```
// Define a function
const fn = function () {
  //...
}; // <-- semicolon added

// then execute some code inside a closure
(function () {
  //...
})();
```

Or

```
var fn = function () {
  //...
}(function () { // <-- start function call arguments on same line
  //...
})();
```

Similarly, tagged templates allow for advanced forms of string interpolation by evaluating the tag as a function to call, passing the template literal elements as arguments.

```
const foo = function() {
  return 'foo';
}

`bar`; // Noncompliant: `bar` passed as a parameter to function. foo is a string, not a function
```

Therefore, the rule also verifies that template literals don’t start on a separate line.

```
function foo() {  // <-- Use a function declaration
  return 'foo';
}

`bar`;
```

Or

```
const foo = function() {
  return 'foo';
}`bar`; // <-- start template literal on same line
```

Resouces

Documentation

- MDN web docs - Automatic semicolon insertion
- MDN web docs - Grouping operator and automatic semicolon insertion
- MDN web docs - Tagged templates

"switch" statements should not have too many "case" clauses

Clave: javascript:S1479

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When `switch` statements have large sets of `case` clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead.

Unused local variables and functions should be removed

Clave: javascript:S1481

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Usually, the fix for this issue is straightforward, you just need to remove the unused variable declaration, or its name from the declaration statement if it is declared along with other variables.

Noncompliant code example

```
function numberOfMinutes(hours) {
  var seconds = 0;   // seconds is never used
  return hours * 60;
}
```

Compliant solution

```
function numberOfMinutes(hours) {
  return hours * 60;
}
```

Noncompliant code example

When an array destructuring is used and some element of the array is never referenced, one might simply remove it from the destructuring.

```
const [_, params] = url.split(path);
```

Compliant solution

```
const [, params] = url.split(path);
```

root_cause

If a local variable or a local function is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable or function is used for.

What is the potential impact?

Dead code

An unused variable or local function usually occurs because some logic is no longer required after a code change. In that case, such code becomes unused and never executed.

Also, if you are writing code for the front-end, every unused variable or function remaining in your codebase is just extra bytes you have to send over the wire to your users. Unused code bloats your codebase unnecessarily and impacts the performance of your application.

Wrong logic

It could happen that due to a bad copy-paste or autocompletion, the wrong variable is used, while the right one is only declared. In that case, the unused variable should be used instead of deleted from the codebase.

Memory leaks

Finally, unused functions can also cause memory leaks. For example, an unused function can create a closure over a variable that would otherwise be released to the garbage collector.

```
let theThing = null;
const replaceThing = function () {
  const originalThing = theThing;
  const unused = function () {
    if (originalThing) {
      console.log("hi");
    }
  };
  theThing = {
    longStr: new Array(1000000).join("*"),
    someMethod: function () {
      console.log(someMessage);
    },
  };
};
setInterval(replaceThing, 1000);
```

resources

Documentation

- MDN web docs - Destructuring assignment / Ignoring some values

Articles & blog posts

- Phil Nash, Common TypeScript Issues Nº 3: unused local variables and functions
- David Glasser, An interesting kind of JavaScript memory leak

---

Local variables should not be declared and then immediately returned or thrown

Clave: javascript:S1488

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. This practice can make the code harder to read and understand, as it introduces an extra step that doesn’t add any value. Instead of declaring a variable and then immediately returning or throwing it, it is generally better to return or throw the value directly. This makes the code cleaner, simpler, and easier to understand.

how_to_fix

Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. To fix the issue, return or throw the value directly.

Noncompliant code example

```
function computeDurationInMilliseconds(hours, minutes, seconds) {
  const duration = (((hours * 60) + minutes) * 60 + seconds) * 1000;
  return duration;
}
```

Compliant solution

```
function computeDurationInMilliseconds(hours, minutes, seconds) {
  return (((hours * 60) + minutes) * 60 + seconds) * 1000;
}
```

Noncompliant code example

```
function doSomething() {
  const myError = new Error();
  throw myError;
}
```

Compliant solution

```
function doSomething() {
  throw new Error();
}
```

Functions should not be nested too deeply

Clave: javascript:S2004

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Nested functions refer to the practice of defining a function within another function. These inner functions have access to the variables and parameters of the outer function, creating a closure.

While nesting functions is a common practice in JavaScript, deeply nested functions can make the code harder to read and understand, especially if the functions are long or if there are many levels of nesting.

This can make it difficult for other developers or even yourself to understand and maintain the code.

Noncompliant code example

With the default threshold of 4 levels:

```
function f() {
  function f_inner() {
    function f_inner_inner() {
      function f_inner_inner_inner() {
        function f_inner_inner_inner_inner() { // Noncompliant
        }
      }
    }
  }
}
```

resources

Documentation

- MDN web docs - Nested functions and closures

Values should not be uselessly incremented

Clave: javascript:S2123

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A value that is incremented or decremented and then not stored is at best wasted code and at worst a bug.

Noncompliant code example

```
function pickNumber() {
  let i = 0;
  i = i++; // Noncompliant; i is still zero

  return i++; // Noncompliant; 0 returned
}
```

Compliant solution

```
function pickNumber() {
  let i = 0;
  i++;

  return ++i;
}
```

Return values from functions without side effects should not be ignored

Clave: javascript:S2201

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a function is called, it executes its block of code and uses a `return` statement within the function to specify the value that the function will produce as its result. This returned value can then be used or assigned to a variable in the calling code.

If a function returns a value that is not used or assigned to a variable, it may indicate that the function is not being used correctly or that there is a mistake in the code. This can make the code harder to understand and maintain, and can also lead to errors if the return value is needed later in the code.

Ignoring the return value of a function can be a sign of poor coding practices. It can indicate that the developer did not fully understand the purpose of the function or did not take the time to properly integrate it into the code.

This rule triggers an issue only on a predefined list of methods from built-in objects (`String`, `Number`, `Date`, `Array`, `Math`, and `RegExp`) to prevent generating any false-positives.

```
'hello'.lastIndexOf('e'); // Noncompliant: The return value is lost
```

Ensure that the return value of the function is used by assigning the return value to a variable.

```
let lastIndex = 'hello'.lastIndexOf('e');
```

Or use the value directly as part of an expression.

```
console.log('hello'.lastIndexOf('e'));
```

resources

Documentation

- MDN web docs - `String`
- MDN web docs - `Number`
- MDN web docs - `Date`
- MDN web docs - `Array`
- MDN web docs - `Math`
- MDN web docs - `RegExp`

Wildcard imports should not be used

Clave: javascript:S2208

Severidad: CRITICAL

Estado: ☑ Activa

Descripción: No disponible

root_cause

On the principle that clearer code is better code, you should explicitly `import` the things you want to use in a module. Using `import *` imports everything in the module and risks confusing maintainers. Similarly, `export * from "module";` imports and then re-exports everything in the module and risks confusing not just maintainers but also the module’s users.

Noncompliant code example

```
import * as Imported from "aModule";  // Noncompliant
```

Compliant solution

```
import {aType, aFunction} from "aModule";
```

Parameters should be passed in the correct order

Clave: javascript:S2234

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Functions

root_cause

Consistent naming between arguments and parameters reduces the chances of making mistakes, such as passing the wrong value to a parameter or omitting an argument in a function call. When the names of arguments in a function call match the names of the function parameters, it contributes to clearer, more readable code.

However, when the names match but are passed in a different order than their declaration in the function signature, it may indicate a mistake in the parameter order, likely leading to unexpected results.

```
function divide(dividend, divisor) {
  return dividend / divisor;
}

function doTheThing() {
  const dividend = 15;
  const divisor = 5;

  const result = divide(divisor, dividend); // Noncompliant: not the expected result
  //...
}
```

Ensure that the arguments passed to the function are in the correct order, according to the function signature.

```
function divide(dividend, divisor) {
  return dividend / divisor;
}

function doTheThing() {
  const dividend = 15;
  const divisor = 5;

  const result = divide(dividend, divisor);
  //...
}
```

Exceptions

Swapped arguments that are compared beforehand in an enclosing `if` statement are ignored:

```
function divide(dividend, divisor) {
  return dividend / divisor;
}

function doTheThing() {
  const dividend = 5;
  const divisor = 15;
  if (divisor > dividend) {
    const result = divide(divisor, dividend);
    //...
  }
}
```

Using pseudorandom number generators (PRNGs) is security-sensitive

Clave: javascript:S2245

Severidad: CRITICAL

Estado: ☑ Activa

Descripción: No disponible

default

PRNGs are algorithms that produce sequences of numbers that only approximate true randomness. While they are suitable for applications like simulations or modeling, they are not appropriate for security-sensitive contexts because their outputs can be predictable if the internal state is known.

In contrast, cryptographically secure pseudorandom number generators (CSPRNGs) are designed to be secure against prediction attacks. CSPRNGs use cryptographic algorithms to ensure that the generated sequences are not only random but also unpredictable, even if part of the sequence or the internal state becomes known. This unpredictability is crucial for security-related tasks such as generating encryption keys, tokens, or any other values that must remain confidential and resistant to guessing attacks.

For example, the use of non-cryptographic PRNGs has led to vulnerabilities such as:

- CVE-2013-6386
- CVE-2006-3419
- CVE-2008-4102

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. Therefore, it is critical to use CSPRNGs in any security-sensitive application to ensure the robustness and security of the system.

As the `Math.random()` function relies on a weak pseudorandom number generator, this function should not be used for security-critical applications or for protecting sensitive data. In such context, a cryptographically strong pseudorandom number generator (CSPRNG) should be used instead.

Ask Yourself Whether

- the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed.
- the function you use is a non-cryptographic PRNG.
- the generated value is used multiple times.
- an attacker can access the generated value.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use a cryptographically secure pseudorandom number generator (CSPRNG) like `crypto.getRandomValues()`.
- Use the generated random values only once.
- You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.

Sensitive Code Example

```
const val = Math.random(); // Sensitive
// Check if val is used in a security context.
```

Compliant Solution

```
// === Client side ===
const crypto = window.crypto || window.msCrypto;
var array = new Uint32Array(1);
crypto.getRandomValues(array);

// === Server side ===
const crypto = require('crypto');
const buf = crypto.randomBytes(1);
```

See

- OWASP - Secure Random Number Generation Cheat Sheet
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

- CWE - CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)
- CWE - CWE-330 - Use of Insufficiently Random Values
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-1241 - Use of Predictable Algorithm in Random Number Generator

assess_the_problem

Ask Yourself Whether

- the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed.
- the function you use is a non-cryptographic PRNG.
- the generated value is used multiple times.
- an attacker can access the generated value.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const val = Math.random(); // Sensitive
// Check if val is used in a security context.
```

root_cause

PRNGs are algorithms that produce sequences of numbers that only approximate true randomness. While they are suitable for applications like simulations or modeling, they are not appropriate for security-sensitive contexts because their outputs can be predictable if the internal state is known.

In contrast, cryptographically secure pseudorandom number generators (CSPRNGs) are designed to be secure against prediction attacks. CSPRNGs use cryptographic algorithms to ensure that the generated sequences are not only random but also unpredictable, even if part of the sequence or the internal state becomes known. This unpredictability is crucial for security-related tasks such as generating encryption keys, tokens, or any other values that must remain confidential and resistant to guessing attacks.

For example, the use of non-cryptographic PRNGs has led to vulnerabilities such as:

- CVE-2013-6386
- CVE-2006-3419
- CVE-2008-4102

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. Therefore, it is critical to use CSPRNGs in any security-sensitive application to ensure the robustness and security of the system.

As the `Math.random()` function relies on a weak pseudorandom number generator, this function should not be used for security-critical applications or for protecting sensitive data. In such context, a cryptographically strong pseudorandom number generator (CSPRNG) should be used instead.

how_to_fix

Recommended Secure Coding Practices

- Use a cryptographically secure pseudorandom number generator (CSPRNG) like `crypto.getRandomValues()`.
- Use the generated random values only once.
- You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.

Compliant Solution

```
// === Client side ===
const crypto = window.crypto || window.msCrypto;
var array = new Uint32Array(1);
crypto.getRandomValues(array);

// === Server side ===
const crypto = require('crypto');
const buf = crypto.randomBytes(1);
```

See

- OWASP - Secure Random Number Generation Cheat Sheet
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)
- CWE - CWE-330 - Use of Insufficiently Random Values
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-1241 - Use of Predictable Algorithm in Random Number Generator

JavaScript parser failure

Clave: javascript:S2260

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When the JavaScript parser fails, it is possible to record the failure as a violation on the file. This way, not only it is possible to track the number of files that do not parse but also to easily find out why they do not parse.

Variables should be used in the blocks where they are declared

Clave: javascript:S2392

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `var` statement declares variables that are function-scoped or globally-scoped. `var` declarations are hoisted, meaning declaring a variable anywhere in the code is equivalent to declaring it at the top of the function or the script.

Even if hoisted, it is still recommended to declare the variable inside the block it is used. This improves readability and maintainability because it makes it clear where the variable is being used. The code then becomes easier to understand and follow, especially for other developers who may be working on the same codebase.

```
function doSomething(a, b) {
  if (a > b) {
    var x = a - b;  // Noncompliant: 'x' is used later outside this block
  }

  if (a > 4) {
    console.log(x);
  }

  for (var i = 0; i < m; i++) { // Noncompliant: both loops use same variable
  }

  for (var i = 0; i < n; i++) {
  }

  return a + b;
}
```

When `var` declaration is used outside of a block, the declaration should be done at the uppermost level where it is used. When possible, use `let` or `const`, which allow for block-scoped variables.

```
function doSomething(a, b) {
  var x;

  if (a > b) {
    x = a - b;
  }

  if (a > 4) {
    console.log(x);
  }

  for (let i = 0; i < m; i++) {
  }

  for (let i = 0; i < n; i++) {
  }


  return a + b;
}
```

resources

Documentation

- MDN web docs - Scope
- MDN web docs - `var`
- MDN web docs - Hoisting

- MDN web docs - const
- MDN web docs - let

Exceptions should not be ignored

Clave: javascript:S2486

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- OWASP - Top 10 2017 Category A10 - Insufficient Logging & Monitoring
- CWE - CWE-390 - Detection of Error Condition Without Action

root_cause

When exceptions occur, it is usually a bad idea to simply ignore them. Instead, it is better to handle them properly, or at least to log them.

Noncompliant code example

```
function f() {
  try {
    doSomething();
  } catch (err) {
  }
}
```

Compliant solution

```
function f() {
  try {
    doSomething();
  } catch (err) {
    console.log(`Exception while doing something: ${err}`);
  }
}
```

File uploads should be restricted

Clave: javascript:S2598

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Noncompliant code example

The following code sample is vulnerable because it implicitly uses `/tmp` or `/var/tmp` as upload directory.

```
const crypto = require('node:crypto');
const multer = require('multer');

let diskStorage = multer.diskStorage({
  filename: (req, file, cb) => {
    const buf = crypto.randomBytes(20);
    cb(null, buf.toString('hex'))
  }
}); // Noncompliant

let diskUpload = multer({
  storage: diskStorage,
});
```

Compliant code example

```
const multer = require('multer');

let diskStorage = multer.diskStorage({
  filename: (req, file, cb) => {
    const buf = crypto.randomBytes(20);
    cb(null, buf.toString('hex'))
  },
  destination: (req, file, cb) => {
    cb(null, '/uploads/')
  }
});

let diskUpload = multer({
  storage: diskStorage,
});
```

How does this work?

Use pre-approved folders

Create a special folder where untrusted data should be stored. This folder should be classified as untrusted and have the following characteristics:

- It should have specific read and write permissions that belong to the right people or organizations.
- It should have a size limit or its size should be monitored.
- It should contain backup copies if it contains data that belongs to users.

This folder should not be located in `/tmp`, `/var/tmp` or in the Windows directory `%TEMP%`.
These folders are usually "world-writable", can be manipulated, and can be accidentally deleted by the system.

Also, the original file names and extensions should be changed to controlled strings to prevent unwanted code from being executed based on the file names.

root_cause

If the file upload feature is implemented without proper folder restriction, it will result in an implicit trust violation within the server, as trusted files will be implicitly stored alongside third-party files that should be considered untrusted.

This can allow an attacker to disrupt the security of an internal server process or the running application.

What is the potential impact?

After discovering this vulnerability, attackers may attempt to upload as many different file types as possible, such as javascript files, bash scripts, malware, or malicious configuration files targeting potential processes.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Full application compromise

In the worst-case scenario, the attackers succeed in uploading a file recognized by in an internal tool, triggering code execution.

Depending on the attacker, code execution can be used with different intentions:

- Download the internal server’s data, most likely to sell it.
- Modify data, install malware, for instance, malware that mines cryptocurrencies.
- Stop services or exhaust resources, for instance, with fork bombs.

Server Resource Exhaustion

By repeatedly uploading large files, an attacker can consume excessive server resources, resulting in a denial of service.

If the component affected by this vulnerability is not a bottleneck that acts as a single point of failure (SPOF) within the application, the denial of service can only affect the attacker who caused it.

Even though a denial of service might have little direct impact, it can have secondary impact in architectures that use containers and container orchestrators. For example, it can cause unexpected container failures or overuse of resources.

In some cases, it is also possible to force the product to "fail open" when resources are exhausted, which means that some security features are disabled in an emergency.

These threats are particularly insidious if the attacked organization does not maintain a disaster recovery plan (DRP).

how_to_fix

Noncompliant code example

```
const Formidable = require('formidable');

const form          = new Formidable(); // Noncompliant
form.uploadDir      = "/tmp/";
form.keepExtensions = true;
```

Compliant solution

```
const Formidable = require('formidable');
const form          = new Formidable();
form.uploadDir      = "/uploads/";
form.keepExtensions = false;
```

How does this work?

Use pre-approved folders

Create a special folder where untrusted data should be stored. This folder should be classified as untrusted and have the following characteristics:

- It should have specific read and write permissions that belong to the right people or organizations.
- It should have a size limit or its size should be monitored.
- It should contain backup copies if it contains data that belongs to users.

This folder should not be located in `/tmp`, `/var/tmp` or in the Windows directory `%TEMP%`.
These folders are usually "world-writable", can be manipulated, and can be accidentally deleted by the system.

Also, the original file names and extensions should be changed to controlled strings to prevent unwanted code from being executed based on the file names.

resources

- OWASP - Top 10 2021 Category A4 - Insecure Design
- CWE - CWE-434 - Unrestricted Upload of File with Dangerous Type
- CWE - CWE-400 - Uncontrolled Resource Consumption
- OWASP Unrestricted File Upload - Unrestricted File Upload

Multiline blocks should be enclosed in curly braces

Clave: javascript:S2681

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Having inconsistent indentation and omitting curly braces from a control structure, such as an `if` statement or `for` loop, is misleading and can induce bugs.

This rule raises an issue when the indentation of the lines after a control structure indicates an intent to include those lines in the block, but the omission of curly braces means the lines will be unconditionally executed once.

The following patterns are recognized:

```
if (condition)
  firstActionInBlock();
  secondAction();  // Noncompliant: secondAction is executed unconditionally
thirdAction();
```

```
if (condition) firstActionInBlock(); secondAction();  // Noncompliant: secondAction is executed unconditionally
```

```
if (condition) firstActionInBlock();  // Noncompliant
  secondAction();  // Executed unconditionally
```

```
if (condition); secondAction();  // Noncompliant: secondAction is executed unconditionally
```

```
let str = undefined;
for (let i = 0; i < array.length; i++)
  str = array[i];
  doTheThing(str);  // Noncompliant: executed only on the last element
```

Note that this rule considers tab characters to be equivalent to 1 space. When mixing spaces and tabs, a code may look fine in one editor but be confusing in another configured differently.

resources

- CWE - CWE-483 - Incorrect Block Delimitation

"arguments.caller" and "arguments.callee" should not be used

Clave: javascript:S2685

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - The arguments object
- MDN web docs - arguments.callee
- MDN web docs - Function.prototype.caller
- MDN web docs - Function.prototype.arguments
- MDN web docs - ReferenceError: deprecated caller or arguments usage

root_cause

In JavaScript, `arguments` is a built-in array-like object automatically available within the scope of all non-arrow functions. It allows you to access the arguments the function was called with, even if the number of arguments passed during the function call does not match the number declared in the function signature. `arguments` has entries for each argument, with the first entry's index at `0`.

The arguments object has two deprecated properties called `arguments.caller` and `arguments.callee`, which were used to refer to functions involved in the function invocation chain:

- The `arguments.callee` property contains the currently executing function that the arguments belong to.
- The `arguments.caller` property returns the function that invoked the currently executing function. It was replaced by `Function.prototype.caller`, which provides the same functionality.

Both `arguments.caller` and `arguments.callee` are non-standard, deprecated, and leak stack information, which poses security risks and severely limits the possibility of optimizations.

Accessing `arguments.callee`, `Function.prototype.caller` and `Function.prototype.arguments` in strict mode will throw a `TypeError`.

```
function whoCalled() {
   if (arguments.caller == null)   //Noncompliant
      console.log('I was called from the global scope.');
   else
      console.log(arguments.caller + ' called me!');  // Noncompliant

  console.log(whoCalled.caller);  // Noncompliant
  console.log(whoCalled.arguments);  // Noncompliant
}
```

"NaN" should not be used in comparisons

Clave: javascript:S2688

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `NaN`
- MDN web docs - `isNaN()`
- MDN web docs - `Number.isNaN()`
- MDN web docs - Type conversion

root_cause

In JavaScript, `NaN` stands for "Not-a-Number." It is a special value representing a numeric data type that is not a valid number. `NaN` is returned as a result when an arithmetic operation or mathematical function is performed, and the result is undefined or unrepresentable as a valid number.

Comparing a value with `NaN` in JavaScript can be problematic because of the way `NaN` behaves in comparison operations. The reason is that `NaN` is not equal to any value, including itself, and this behavior can lead to unexpected results.

```
const a = NaN;

if (a === NaN) { // Noncompliant: Always false
  console.log("a is not a number"); // This is dead code
}

if (a !== NaN) { // Noncompliant: Always true
  console.log("a is not NaN"); // This statement is not necessarily true
}
```

To check if a value is `NaN`, you should use the `isNaN()` function:

```
const a = NaN;

if (isNaN(a)) {
  console.log("a is not a number");
}

if (!isNaN(a)) {
  console.log("a is not NaN");
}
```

Keep in mind that `isNaN()` can be a bit quirky since it tries to convert its argument into a number before checking if it is `NaN`. If the argument cannot be converted into a number, `isNaN()` will return true, which may not be the desired behavior in all cases.

Instead, you should prefer using the `Number.isNaN()` method over `isNaN()` to perform a strict check for `NaN` without any type conversion:

```
const a = NaN;

if (Number.isNaN(a)) {
  console.log("a is not a number");
}

if (!Number.isNaN(a)) {
  console.log("a is not NaN");
}
```

"indexOf" checks should not be for positive numbers

Clave: javascript:S2692

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `Array.prototype.includes()`

root_cause

Most checks against an `indexOf` value compare it with -1 because 0 is a valid index. Checking against `> 0` ignores the first element, which is likely a bug.

```
let arr = ["blue", "red"];

if (arr.indexOf("blue") > 0) { // Noncompliant
  // ...
}
```

Moreover, if the intent is merely to check the presence of the element, and if your browser version supports it, consider using `includes` instead.

```
let arr = ["blue", "red"];

if (arr.includes("blue")) {
  // ...
}
```

This rule raises an issue when an `indexOf` value retrieved from an array is tested against `> 0`.

Tests should include assertions

Clave: javascript:S2699

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

An assertion is a statement within the unit test that checks whether a particular condition is true or false. It defines the expected behavior of the unit being tested. Assertions are used to express the test’s expected outcome, and they are the criteria against which the actual output of the unit is evaluated.

When the unit test is executed, the assertions are evaluated. If all the assertions in the test pass, it means the unit is functioning correctly for that specific set of inputs. If any of the assertions fail, it indicates that there is a problem with the unit's implementation, and the test case helps identify the issue.

Without assertions, a unit test doesn't actually verify anything, making it ineffective in catching potential bugs or regressions. It will always pass, regardless of the implementation of the unit. This can lead to a false sense of security, as you may believe that your code is working correctly when it might not be.

This rule raises an issue when one of the following assertion libraries is imported but no assertion is used in a test:

- `chai`
- `sinon`
- `vitest`
- `supertest`

```
const expect = require("chai").expect;

describe("No assertions", function() {
    it("don't test anything", function() { // Noncompliant: The unit test doesn't assert anything
        const str = "";
    });
});
```

To write effective unit tests, you should define the expected behavior of the unit using assertions, allowing you to catch bugs early and ensure the reliability of your codebase.

```
const expect = require("chai").expect;

describe("Assertions", function() {
    it("test something", function() {
        const str = "";
        expect(str).to.be.a("string");
    });
});
```

resources

Documentation

- Chai.js Documentation - API Reference
- Sinon.js Documentation - Assertions

"delete" should be used only with object properties

Clave: javascript:S3001

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `delete` operator
- MDN web docs - Global object
- MDN web docs - `globalThis`
- MDN web docs - ECMAScript modules
- Node.js Documentation - CommonJS modules

root_cause

The `delete` operator is used to remove a property from an object. It only affects its own properties. There are two valid ways to remove a property:

- Using the dot notation: `delete object.property`
- Using the bracket notation: `delete object[property]`

`delete` will throw a `TypeError` in strict mode if the property is a non-configurable property.

`delete identifier` may work if `identifier` is a **configurable** property of the global object. For `identifier` to be **configurable**, it should have been declared directly as a `globalThis` property (`globalThis.identifier = 1`). This form is not common practice and should be avoided. Use `delete globalThis.identifier` instead if needed.

Aside from that case, deleting variables, including function parameters, never works:

- Variables declared with `var` cannot be deleted from the global or a function's scope, because while they may be attached to the global object, they are **non-configurable**. In CommonJS and ECMAScript modules, top-level variable declarations are scoped to the module and not attached to the global object.

- Variables declared with `let` or `const` are not attached to any object.

```
var x = 1;
delete x; // Noncompliant: depending on the context, this does nothing or throws TypeError

function foo(){}
delete foo; // Noncompliant: depending on the context, this does nothing or throws TypeError
```

Avoid using the `delete identifier` form. Instead, use one of the valid forms.

```
var obj = {
  x: 1,
  foo: function(){
  ...
  }
};
delete obj['x'];
delete obj.foo;
```

Comparison operators should not be used with strings

Clave: javascript:S3003

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The use of comparison operators (`<`, `<=`, `>=`, `>`) with strings is not likely to yield the expected results. Make sure the intention was to compare strings and not numbers.

Noncompliant code example

```
var appleNumber = "123";
var orangeNumber = "45";
if (appleNumber < orangeNumber) {  // Noncompliant, this condition is true
  alert("There are more oranges");
}
```

Compliant solution

```
var appleNumber = "123";
var orangeNumber = "45";
if (Number(appleNumber) < Number(orangeNumber)) {
  alert("There are more oranges");
}
```

Exceptions

The rule ignores string comparisons occurring in the callback of a sort invocation, e.g.:

```
const fruits = ['orange', 'apple', 'banana'];
fruits.sort((a, b) => a < b);
```

Creating cookies without the "HttpOnly" flag is security-sensitive

Clave: javascript:S3330

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

- By default the `HttpOnly` flag should be set to *true* for most of the cookies and it’s mandatory for session / sensitive-security cookies.

Compliant Solution

cookie-session module:

```
let session = cookieSession({
  httpOnly: true,// Compliant
});  // Compliant
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    httpOnly: true // Compliant
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  httpOnly: true // Compliant
}); // Compliant
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { httpOnly: true }}); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP HttpOnly
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-1004 - Sensitive Cookie Without 'HttpOnly' Flag
- Derived from FindSecBugs rule HTTPONLY_COOKIE
- STIG Viewer - Application Security and Development: V-222575 - The application must set the HTTPOnly flag on session cookies.

assess_the_problem

Ask Yourself Whether

- the cookie is sensitive, used to authenticate the user, for instance a *session-cookie*
- the `HttpOnly` attribute offer an additional protection (not the case for an *XSRF-TOKEN cookie* / CSRF token for example)

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

cookie-session module:

```
let session = cookieSession({
  httpOnly: false,// Sensitive
});  // Sensitive
```

express-session module:

```
const express = require('express'),
const session = require('express-session'),

let app = express()
app.use(session({
  cookie:
  {
    httpOnly: false // Sensitive
  }
})),
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  httpOnly: false // Sensitive
}); // Sensitive
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { httpOnly: false }}); // Sensitive
```

root_cause

When a cookie is configured with the `HttpOnly` attribute set to *true*, the browser guaranties that no client-side script will be able to read it. In most cases, when a cookie is created, the default value of `HttpOnly` is *false* and it's up to the developer to decide whether or not the content of the cookie can be read by the client-side script. As a majority of Cross-Site Scripting (XSS) attacks target the theft of session-cookies, the `HttpOnly` attribute can help to reduce their impact as it won't be possible to exploit the XSS vulnerability to steal session-cookies.

default

When a cookie is configured with the `HttpOnly` attribute set to *true*, the browser guaranties that no client-side script will be able to read it. In most cases, when a cookie is created, the default value of `HttpOnly` is *false* and it's up to the developer to decide whether or not the content of the cookie can be read by the client-side script. As a majority of Cross-Site Scripting (XSS) attacks target the theft of session-cookies, the `HttpOnly` attribute can help to reduce their impact as it won't be possible to exploit the XSS vulnerability to steal session-cookies.

Ask Yourself Whether

- the cookie is sensitive, used to authenticate the user, for instance a *session-cookie*
- the `HttpOnly` attribute offer an additional protection (not the case for an *XSRF-TOKEN cookie* / CSRF token for example)

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- By default the `HttpOnly` flag should be set to *true* for most of the cookies and it's mandatory for session / sensitive-security cookies.

Sensitive Code Example

cookie-session module:

```
let session = cookieSession({
  httpOnly: false,// Sensitive
});  // Sensitive
```

express-session module:

```
const express = require('express'),
const session = require('express-session'),

let app = express()
app.use(session({
  cookie:
  {
    httpOnly: false // Sensitive
  }
})),
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  httpOnly: false // Sensitive
}); // Sensitive
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { httpOnly: false }}); // Sensitive
```

Compliant Solution

cookie-session module:

```
let session = cookieSession({
  httpOnly: true,// Compliant
});  // Compliant
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    httpOnly: true // Compliant
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  httpOnly: true // Compliant
}); // Compliant
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { httpOnly: true }}); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP HttpOnly
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-1004 - Sensitive Cookie Without 'HttpOnly' Flag
- Derived from FindSecBugs rule HTTPONLY_COOKIE
- STIG Viewer - Application Security and Development: V-222575 - The application must set the HTTPOnly flag on session cookies.

Unchanged variables should be marked as "const"

Clave: javascript:S3353

Severidad: CRITICAL

Estado: ☑ Activa

Descripción: No disponible

how_to_fix

Mark the given variable with the `const` modifier.

Noncompliant code example

```
function seek(input) {
  let target = 32;  // Noncompliant
  for (const i of input) {
    if (i === target) {
      return true;
    }
  }
  return false;
}
```

Compliant solution

```
function seek(input) {
  const target = 32;
  for (const i of input) {
    if (i === target) {
      return true;
    }
  }
  return false;
}

function getUrl(protocol, domain, path) {
  let url; // Noncompliant
  url = `${protocol}/${domain}/${path}`;
  return url;
}
```

Compliant solution

```
function getUrl(protocol, domain, path) {
  const url = `${protocol}/${domain}/${path}`;
  return url;
}
```

root_cause

If a variable that is not supposed to change is not marked as `const`, it could be accidentally reassigned elsewhere in the code, leading to unexpected behavior and bugs that can be hard to track down.

By declaring a variable as `const`, you ensure that its value remains constant throughout the code. It also signals to other developers that this value is intended to remain constant. This can make the code easier to understand and maintain.

In some cases, using `const` can lead to performance improvements. The compiler might be able to make optimizations knowing that the value of a `const` variable will not change.

resources

Documentation

- MDN - const

Ternary operators should not be nested

Clave: javascript:S3358

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Nested ternaries are hard to read and can make the order of operations complex to understand.

```
function getReadableStatus(job) {
  return job.isRunning() ? "Running" : job.hasErrors() ? "Failed" : "Succeeded ";  // Noncompliant
}
```

Instead, use another line to express the nested operation in a separate statement.

```
function getReadableStatus(job) {
  if (job.isRunning()) {
    return "Running";
  }
  return job.hasErrors() ? "Failed" : "Succeeded";
}
```

Exceptions

This rule does not apply in JSX expressions to support conditional rendering and conditional attributes as long as the nesting happens in separate JSX expression containers, i.e. JSX elements embedding JavaScript code, as shown below:

```
return (
<>
  {isLoading ? (
    <Loader active />
  ) : (
    <Panel label={isEditing ? 'Open' : 'Not open'}>
      <a>{isEditing ? 'Close now' : 'Start now'}</a>
      <Checkbox onClick={!saving ? setSaving(saving => !saving) : null} />
    </Panel>
  )}
</>
);
```

If you have nested ternaries in the same JSX expression container, refactor your logic into a separate function like that:

```
function myComponent(condition) {
  if (condition < 0) {
    return '<DownSign>it is negative</DownSign>';
  } else if (condition > 0) {
    return '<UpSign>it is positive</UpSign>';
  } else {
    return '<BarSign>it is zero</BarSign>';
  }
}

return (
  {myComponent(foo)}
);
```

resources

Articles & blog posts

- Sonar - Stop nesting ternaries in JavaScript

Assertion arguments should be passed in the correct order

Clave: javascript:S3415

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Node.js Documentation - Assert
- Chai.js Documentation - Assert
- Chai.js Documentation - `expect` and `should`

root_cause

Assertions are statements that check whether certain conditions are true. They are used to validate that the actual results of a code snippet match the expected outcomes. By using assertions, developers can ensure that their code behaves as intended and identify potential bugs or issues early in the development process.

The convention for passing assertion arguments is to pass the expected value as the first argument and the actual value as the second argument. This convention is based on the idea that the expected value is what the code is supposed to produce, and the actual value is what the code produces. By passing the expected value first, it is easier to understand the intent of the assertion and to quickly identify any errors that may be present. Additionally, many testing frameworks and libraries expect assertion arguments to be passed in this order, so following the convention can help ensure that your code works correctly with these tools.

What is the potential impact?

Having the expected value and the actual value in the wrong order will not alter the outcome of tests, (succeed/fail when it should) but the error messages will contain misleading information.

This rule raises an issue when the actual argument to an assertions library method is a hard-coded value and the expected argument is not.

how_to_fix

You should provide the assertion methods with a hard-coded value as the expected value, while the actual value of the assertion should derive from the portion of code that you want to test.

Noncompliant code example

```
const assert = require('chai').assert;
const expect = require('chai').expect;
const should = require('chai').should();

it("inverts arguments", function() {
    assert.equal(42, aNumber); // Noncompliant: actual value is passed as first argument and expected as second argument
    expect(42).to.equal(aNumber); // Noncompliant: actual value is passed as first argument and expected as second argument
    should.fail(42, aNumber);  // Noncompliant: actual value is passed as first argument and expected as second argument
});
```

Compliant solution

Swap the order of the assertion arguments so that the expected value is passed as the first argument and the actual value is passed as the second argument.

```
const assert = require('chai').assert;
const expect = require('chai').expect;
const should = require('chai').should();

it("inverts arguments", function() {
    assert.equal(aNumber, 42);
    expect(aNumber).to.equal(42);
    should.fail(aNumber, 42);
});
```

Generators should explicitly "yield" a value

Clave: javascript:S3531

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Generator
- MDN web docs - `yield`

root_cause

In JavaScript, a generator is a special type of function that can be paused and resumed during its execution. It allows you to define an iterative algorithm by writing a function that can maintain its internal state and produce a sequence of values over time.

Generators are defined using a function syntax with an asterisk (`*`) appended to the `function` keyword (`function*`). Within the generator function, you can use the `yield` keyword to produce a value and temporarily pause the execution of the function, returning that value to the consumer.

```
function* generate() {
  yield 1;
  yield 2;
  yield 3;
}
```

This example defines a generator function named `generate` that produces a sequence of values: 1, 2, and 3.

Using a generator without the `yield` keyword can limit the usefulness and potential benefits of generators. When you use the `yield` keyword without providing a value, it creates a yield expression that pauses the execution of the generator function and returns `undefined` as the yielded value.

```
function* range(start, end) {
  while (start < end) {
    yield; // Noncompliant: The generator yields undefined
    start++;
  }
}
```

Yielding without a value makes it harder for the generator consumer to understand the purpose or context of the yielded value. Instead, one should always provide an explicit value with `yield` (using `undefined` when that is the intention) to make the generated sequence more meaningful and informative.

```
function* range(start, end) {
  while (start < end) {
    yield start;
    start++;
  }
}
```

"import" should be used to include external code

Clave: javascript:S3533

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Before ECMAScript 2015, module management had to be ad-hoc or provided by 3rd-party libraries such as Node.js, Webpack, or RequireJS. Fortunately, ES2015, provides language-standard mechanisms for module management, `import` and `export`, and older usages should be converted.

Noncompliant code example

```
// circle.js
exports.area = function (r) {
  return PI * r * r;
};

// foo.js
define(["./cart", "./horse"], function(cart, horse) {  // Noncompliant
  // ...
});

// bar.js
const circle = require('./circle.js');  // Noncompliant
```

Compliant solution

```
// circle.js
let area = function (r) {
  return PI * r * r;
}
export default area;
```

```
// foo.js
import cart from "./cart.js";
import horse from "./horse.js";

// bar.js
import circle from "./circle.js"
```

Array indexes should be numeric

Clave: javascript:S3579

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, array indexes should be numeric because arrays are implemented as objects with numeric keys. When an array is created, it is actually an object with properties that are numeric keys. These keys are used to access and assign values stored in the array.

If an array index is not numeric, it will be converted to a string and used as a property name. This can lead to unexpected behavior, as properties are not guaranteed to be ordered in the same way as numeric indexes. For example, if an array has both numeric and non-numeric keys, the non-numeric keys may appear in a different order than the numeric keys.

```
let arr = [];
arr[0] = 'a';
arr['name'] = 'bob'; // Noncompliant: The 'name' property with value 'bob' appears after the elements 'a' and 'foo'
arr[1] = 'foo';
```

Using numeric indexes helps maintain consistency and clarity in your code. It follows the common convention of using numbers to represent positions or indices within an ordered sequence. This makes it easier for developers to understand and work with arrays. Furthermore, JavaScript engines can optimize memory allocation and retrieval of array elements in constant time.

```
let arr = [];
arr[0] = 'a';
arr[1] = 'foo';
arr[2] = 'bob';
```

If you really need to use arrays with additional properties, you should wrap the array into another object, add new properties to that object, and preserve the ordered nature of the wrapped array.

```
let obj = {
  name: 'bob',
  arr: ['a', 'foo']
};
```

resources

Documentation

- MDN web docs - Array
- MDN web docs - Array indices
- MDN web docs - Object

Functions should be called consistently with or without "new"

Clave: javascript:S3686

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `new` operator

root_cause

JavaScript has the `new` keyword that is used in conjunction with constructor functions to create new instances of objects. When you use the `new` keyword with a function, it signifies that the function is intended to be used as a constructor function to create objects.

Any function can be used as a constructor function by convention. Constructor functions are used to create new objects with the same structure or properties. They are typically named with an initial capital letter to distinguish them from regular functions.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

To create a new instance of an object using the constructor function, you use the `new` keyword before the function call.

```
const person1 = new Person('Alice', 30);
const person2 = new Person('Bob', 25);
```

Constructor functions, which create new object instances, must only be called with `new`. Non-constructor functions must not. Mixing these two usages could lead to unexpected results at runtime.

```
function getNum() {
  return 5;
}

function Num(numeric, alphabetic) {
  this.numeric = numeric;
  this.alphabetic = alphabetic;
}

const num1 = getNum();
const num2 = new getNum(); // Noncompliant: An empty object is returned, not 5

const obj1 = new Num();
const obj2 = Num(); // Noncompliant: undefined is returned, not an object
```

The rule checks that the `new` operator is consistently used with each function’s invocations, meaning for all invocations or none.

Literals should not be thrown

Clave: javascript:S3696

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, throwing literals (primitive values like strings, numbers, booleans, etc.) as exceptions is generally discouraged. While it is syntactically valid to throw literals, it is considered a best practice to throw instances of the `Error` class or its subclasses instead.

Throwing an instance of the `Error` class allows you to provide more meaningful information about the error.

The `Error` class and its subclasses provide properties like `message` and `stack` that can be used to convey useful details about the error, such as a description of the problem, the context in which it occurred, or a stack trace for debugging.

```
throw 404;                              // Noncompliant
throw "Invalid negative index.";        // Noncompliant
```

Throwing literals can make it harder to handle and differentiate between different types of errors. Instead, you should use one of the exception types specifically created for the purpose or define your own subclass of the `Error` class.

```
throw new Error("Status: " + 404);
throw new RangeError("Invalid negative index.");
```

resources

Documentation

- MDN web docs - `throw`
- MDN web docs - Error

The return value of void functions should not be used

Clave: javascript:S3699

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `undefined` global property
- MDN web docs - `return` statement

root_cause

When a function in JavaScript does not have a return statement or if it has a return statement without a value, it implicitly returns `undefined`. This means that a function without a return statement or with an empty return statement is, in a way, a "void" function, as it doesn’t return any specific value.

Therefore, attempting to use the return value of a void function in JavaScript is meaningless, and it can lead to unexpected behavior or errors.

Noncompliant code example

```
function foo() {
  console.log("Hello, World!");
}

let a = foo(); // Noncompliant: Assigning the return value of a void function
```

You should not use in any way the return value of a void function.

Compliant solution

```
function foo() {
  console.log("Hello, World!");
}

foo();
```

introduction

This rule raises an issue when a function call result is used, even though the function does not return anything.

Cognitive Complexity of functions should not be too high

Clave: javascript:S3776

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Reducing cognitive complexity can be challenging.
Here are a few suggestions:

- **Extract complex conditions in a new function.**
  Mixed operators in condition will increase complexity. Extracting the condition in a new function with an appropriate name will reduce cognitive load.
- **Break down large functions.**
  Large functions can be hard to understand and maintain. If a function is doing too many things, consider breaking it down into smaller, more manageable functions. Each function should have a single responsibility.
- **Avoid deep nesting by returning early.**
  To avoid the nesting of conditions, process exceptional cases first and return early.
- **Use null-safe operations (if available in the language).**
  When available the `.?` or `??` operator replaces multiple tests and simplifies the flow.

Extraction of a complex condition in a new function.

Noncompliant code example

The code is using a complex condition and has a cognitive cost of 3.

```
function calculateFinalPrice(user, cart) {
  let total = calculateTotal(cart);
  if (user.hasMembership                       // +1 (if)
    && user.orders > 10                        // +1 (more than one condition)
    && user.accountActive
    && !user.hasDiscount
    || user.orders === 1) {                    // +1 (change of operator in condition)
      total = applyDiscount(user, total);
  }
  return total;
}
```

Compliant solution

Even if the cognitive complexity of the whole program did not change, it is easier for a reader to understand the code of the `calculateFinalPrice` function, which now only has a cognitive cost of 1.

```
function calculateFinalPrice(user, cart) {
  let total = calculateTotal(cart);
  if (isEligibleForDiscount(user)) {       // +1 (if)
    total = applyDiscount(user, total);
  }
  return total;
}

function isEligibleForDiscount(user) {
  return user.hasMembership
    && user.orders > 10                     // +1 (more than one condition)
    && user.accountActive
    && !user.hasDiscount
    || user.orders === 1                    // +1 (change of operator in condition)
}
```

Break down large functions.

Noncompliant code example

For example, consider a function that calculates the total price of a shopping cart, including sales tax and shipping.
Note: The code is simplified here, to illustrate the purpose. Please imagine there is more happening in the `for` loops.

```
function calculateTotal(cart) {
  let total = 0;
  for (let i = 0; i < cart.length; i++) {       // +1 (for)
    total += cart[i].price;
  }

  // calculateSalesTax
  for (let i = 0; i < cart.length; i++) {       // +1 (for)
    total += 0.2 * cart[i].price;
  }

  //calculateShipping
  total += 5 * cart.length;

  return total;
}
```

This function could be refactored into smaller functions: The complexity is spread over multiple functions and the complex `calculateTotal` has now a complexity score of zero.

Compliant solution

```
function calculateTotal(cart) {
  let total = calculateSubtotal(cart);
  total += calculateSalesTax(cart);
  total += calculateShipping(cart);
  return total;
}

function calculateSubtotal(cart) {
  let subTotal = 0;
  for (const item of cart) {        // +1 (for)
    subTotal += item.price;
  }
  return subTotal;
}

function calculateSalesTax(cart) {
  let salesTax = 0;
  for (const item of cart) {        // +1 (for)
    salesTax += 0.2 * item.price;
  }
  return salesTax;
}

function calculateShipping(cart) {
  return 5 * cart.length;
}
```

Avoid deep nesting by returning early.

Noncompliant code example

The below code has a cognitive complexity of 6.

```
function calculateDiscount(price, user) {
  if (isEligibleForDiscount(user)) {  // +1 ( if )
```

```
    if (user?.hasMembership) {        // +2 ( nested if )
      return price * 0.9;
  } else if (user?.orders === 1 ) {   // +1 ( else )
          return price * 0.95;
    } else {                          // +1 ( else )
      return price;
    }
  } else {                            // +1 ( else )
    return price;
  }
}
```

Compliant solution

Checking for the edge case first flattens the `if` statements and reduces the cognitive complexity to 3.

```
function calculateDiscount(price, user) {
    if (!isEligibleForDiscount(user)) {  // +1 ( if )
      return price;
    }
    if (user?.hasMembership) {           // +1 ( if )
      return price * 0.9;
    }
    if (user?.orders === 1) {            // +1 ( if )
      return price * 0.95;
    }
    return price;
}
```

Use the optional chaining operator to access data.

In the below code, the cognitive complexity is increased due to the multiple checks required to access the manufacturer’s name. This can be simplified using the optional chaining operator.

Noncompliant code example

```
let manufacturerName = null;

if (product && product.details && product.details.manufacturer) { // +1 (if) +1 (multiple condition)
    manufacturerName = product.details.manufacturer.name;
}
if (manufacturerName) { // +1 (if)
  console.log(manufacturerName);
} else {
  console.log('Manufacturer name not found');
}
```

Compliant solution

The optional chaining operator will return `undefined` if any reference in the chain is `undefined` or `null`, avoiding multiple checks:

```
let manufacturerName = product?.details?.manufacturer?.name;

if (manufacturerName) { // +1 (if)
  console.log(manufacturerName);
} else {
  console.log('Manufacturer name not found');
}
```

Pitfalls

As this code is complex, ensure that you have unit tests that cover the code before refactoring.

introduction

This rule raises an issue when the code cognitive complexity of a function is above a certain threshold.

root_cause

Cognitive Complexity is a measure of how hard it is to understand the control flow of a unit of code. Code with high cognitive complexity is hard to read, understand, test, and modify.

As a rule of thumb, high cognitive complexity is a sign that the code should be refactored into smaller, easier-to-manage pieces.

Which syntax in code does impact cognitive complexity score?

Here are the core concepts:

- **Cognitive complexity is incremented each time the code breaks the normal linear reading flow.**
  This concerns, for example, loop structures, conditionals, catches, switches, jumps to labels, and conditions mixing multiple operators.
- **Each nesting level increases complexity.**
  During code reading, the deeper you go through nested layers, the harder it becomes to keep the context in mind.

- **Method calls are free**
  A well-picked method name is a summary of multiple lines of code. A reader can first explore a high-level view of what the code is performing then go deeper and deeper by looking at called functions content.
  Note: This does not apply to recursive calls, those will increment cognitive score.

The method of computation is fully detailed in the pdf linked in the resources.

Note that the calculation of cognitive complexity at function level deviates from the documented process. Given the functional nature of JavaScript, nesting functions is a prevalent practice, especially within frameworks like React.js. Consequently, the cognitive complexity of functions remains independent from one another. This means that the complexity of a nesting function does not increase with the complexity of nested functions.

What is the potential impact?

Developers spend more time reading and understanding code than writing it. High cognitive complexity slows down changes and increases the cost of maintenance.

Exceptions

Cognitive complexity calculations exclude logical expressions using the `||` and `??` operators.

```
function greet(name) {
    name = name || 'Guest';
    console.log('Hello, ' + name + '!');
}
```

resources

Documentation

- Sonar - Cognitive Complexity

Articles & blog posts

- Sonar Blog - 5 Clean Code Tips for Reducing Cognitive Complexity

---

Arguments to built-in functions should match documented types

Clave: javascript:S3782

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Global Objects
- MDN web docs - Type coercion

root_cause

The types of the arguments that built-in functions accept are specified in the JavaScript language specification. Calls to these functions should conform to the documented types as they are designed to work with specific data types. If the arguments passed to these functions do not match the expected types, it can lead to type errors or unexpected behavior.

Additionally, passing the correct types of arguments to built-in functions can improve performance by reducing the need for type conversions and other operations.

```
const isTooSmall = Math.abs(x < 0.0042); // Noncompliant: 'Math.abs' takes a number as argument
```

Ensure that the arguments passed to built-in functions match the documented types. This is an important aspect of writing high-quality, maintainable, and performant code. You can refer to the Mozilla Developer Network (MDN) documentation for the built-in functions. The documentation typically includes information about the expected types of arguments, the return type of the function, and any other relevant details.

```
const isTooSmall = Math.abs(x) < 0.0042;
```

"in" should not be used with primitive types

Clave: javascript:S3785

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - in operator
- MDN web docs - Primitive
- MDN web docs - Number() constructor
- MDN web docs - String() constructor
- MDN web docs - Boolean() constructor

root_cause

In JavaScript, the `in` operator is primarily used to check if a property exists in an object or if an index exists in an array. However, it is not suitable for use with primitive types such as numbers, strings, or booleans because they are not objects and do not have properties.

If the right operand is of primitive type, the `in` operator raises a `TypeError`.

```
let x = "Foo";
"length" in x; // Noncompliant: TypeError
0 in x;        // Noncompliant: TypeError
"foobar" in x; // Noncompliant: TypeError
```

You should use the object equivalents of numbers, strings, or booleans if you really want to check property existence with the `in` operator.

```
let x = new String("Foo");
"length" in x;    // true
0 in x;           // true
"foobar" in x;    // false
```

Template literal placeholder syntax should not be used in regular strings

Clave: javascript:S3786

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Template strings allow developers to embed variables or expressions in strings using template literals, instead of string concatenation. This is done by using expressions like `${variable}` in a string between two back-ticks (`` ` ``). However, when used in a regular string literal (between double or single quotes) the template will not be evaluated and will be used as a literal, which is probably not what was intended.

Noncompliant code example

```
console.log("Today is ${date}"); // Noncompliant
```

Compliant solution

```
console.log(`Today is ${date}`);
```

"await" should not be used redundantly

Clave: javascript:S4326

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

An `async` function always wraps the return value in a `Promise`. Using `return await` is not required and comes at an extra performance cost. However, you might wish to keep it as it preserves the function call in the stack trace in case an error is thrown asynchronously.

Noncompliant code example

```
async function foo() {
  // ...
}
```

```
async function bar() {
  // ...
  return await foo(); // Noncompliant
}
```

Compliant solution

```
async function foo() {
  // ...
}

async function bar() {
  // ...
  return foo();
}
```

Weak SSL/TLS protocols should not be used

Clave: javascript:S4423

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Articles & blog posts

- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm

introduction

This vulnerability exposes encrypted data to a number of attacks whose goal is to recover the plaintext.

how_to_fix

Noncompliant code example

NodeJs offers multiple ways to set weak TLS protocols. For https and tls, these options are used and are used in other third-party libraries as well.

The first is `secureProtocol`:

```
const https = require('node:https');
const tls   = require('node:tls');

let options = {
 secureProtocol: 'TLSv1_method' // Noncompliant
};

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

The second is the combination of `minVersion` and `maxVerison`. Note that they cannot be specified along with the `secureProtocol` option:

```
const https = require('node:https');
const tls   = require('node:tls');

let options = {
  minVersion: 'TLSv1.1',  // Noncompliant
  maxVersion: 'TLSv1.2'
};

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

And `secureOptions`, which in this example instructs the OpenSSL protocol to turn off some algorithms altogether. In general, this option might trigger side effects and should be used carefully, if used at all.

```
const https     = require('node:https');
const tls       = require('node:tls');
const constants = require('node:crypto'):

let options = {
  secureOptions:
    constants.SSL_OP_NO_SSLv2
    | constants.SSL_OP_NO_SSLv3
    | constants.SSL_OP_NO_TLSv1
}; // Noncompliant

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

Compliant solution

```
const https = require('node:https');
const tls   = require('node:tls');

let options = {
  secureProtocol: 'TLSv1_2_method'
};

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

```
const https = require('node:https');
const tls   = require('node:tls');

let options = {
  minVersion: 'TLSv1.2',
  maxVersion: 'TLSv1.2'
};

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

Here, the goal is to turn on only TLSv1.2 and higher, by turning off all lower versions:

```
const https = require('node:https');
const tls   = require('node:tls');

let options = {
  secureOptions:
    constants.SSL_OP_NO_SSLv2
    | constants.SSL_OP_NO_SSLv3
    | constants.SSL_OP_NO_TLSv1
    | constants.SSL_OP_NO_TLSv1_1
};

let req    = https.request(options, (res) => { });
let socket = tls.connect(443, "www.example.com", options, () => { });
```

How does this work?

As a rule of thumb, by default you should use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

The best choices at the moment are the following.

Use TLS v1.2 or TLS v1.3

Even though TLS V1.3 is available, using TLS v1.2 is still considered good and secure practice by the cryptography community.

The use of TLS v1.2 ensures compatibility with a wide range of platforms and enables seamless communication between different systems that do not yet have TLS v1.3 support.

The only drawback depends on whether the framework used is outdated: its TLS v1.2 settings may enable older and insecure cipher suites that are deprecated as insecure.

On the other hand, TLS v1.3 removes support for older and weaker cryptographic algorithms, eliminates known vulnerabilities from previous TLS versions, and improves performance.

how_to_fix

Noncompliant code example

```
import { aws_apigateway as agw } from 'aws-cdk-lib';
new agw.DomainName(this, 'Example', {
    certificate: certificate,
    domainName: domainName,
    securityPolicy: agw.SecurityPolicy.TLS_1_0, // Noncompliant
});
```

The resource `CfnDomain` uses a weak TLS security policy, by default.

```
import { aws_opensearchservice as os } from 'aws-cdk-lib';

new os.CfnDomain(this, 'Example', {
  domainEndpointOptions: {
    enforceHttps: true,
  },
}); // Noncompliant
```

Compliant solution

```
import { aws_apigateway as agw } from 'aws-cdk-lib';
new agw.DomainName(this, 'Example', {
    certificate: certificate,
    domainName: domainName,
    securityPolicy: agw.SecurityPolicy.TLS_1_2,
});
```

```
import { aws_opensearchservice as os } from 'aws-cdk-lib';

new os.CfnDomain(this, 'Example', {
  domainEndpointOptions: {
    enforceHttps: true,
    tlsSecurityPolicy: 'Policy-Min-TLS-1-2-2019-07',
  },
});
```

How does this work?

As a rule of thumb, by default you should use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

The best choices at the moment are the following.

Use TLS v1.2 or TLS v1.3

Even though TLS V1.3 is available, using TLS v1.2 is still considered good and secure practice by the cryptography community.

The use of TLS v1.2 ensures compatibility with a wide range of platforms and enables seamless communication between different systems that do not yet have TLS v1.3 support.

The only drawback depends on whether the framework used is outdated: its TLS v1.2 settings may enable older and insecure cipher suites that are deprecated as insecure.

On the other hand, TLS v1.3 removes support for older and weaker cryptographic algorithms, eliminates known vulnerabilities from previous TLS versions, and improves performance.

root_cause

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

To provide communication security over a network, SSL and TLS are generally used. However, it is important to note that the following protocols are all considered weak by the cryptographic community, and are officially deprecated:

- SSL versions 1.0, 2.0 and 3.0
- TLS versions 1.0 and 1.1

When these unsecured protocols are used, it is best practice to expect a breach: that a user or organization with malicious intent will perform mathematical attacks on this data after obtaining it by other means.

What is the potential impact?

After retrieving encrypted data and performing cryptographic attacks on it on a given timeframe, attackers can recover the plaintext that encryption was supposed to protect.

Depending on the recovered data, the impact may vary.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Additional attack surface

By modifying the plaintext of the encrypted message, an attacker may be able to trigger additional vulnerabilities in the code. An attacker can further exploit a system to obtain more information.
Encrypted values are often considered trustworthy because it would not be possible for a third party to modify them under normal circumstances.

Breach of confidentiality and privacy

When encrypted data contains personal or sensitive information, its retrieval by an attacker can lead to privacy violations, identity theft, financial loss, reputational damage, or unauthorized access to confidential systems.

In this scenario, the company, its employees, users, and partners could be seriously affected.

The impact is twofold, as data breaches and exposure of encrypted data can undermine trust in the organization, as customers, clients and stakeholders may lose confidence in the organization’s ability to protect their sensitive data.

Legal and compliance issues

In many industries and locations, there are legal and compliance requirements to protect sensitive data. If encrypted data is compromised and the plaintext can be recovered, companies face legal consequences, penalties, or violations of privacy laws.

Cryptographic keys should be robust

Clave: javascript:S4426

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

In today’s cryptography, the length of the **key** directly affects the security level of cryptographic algorithms.

Note that depending on the algorithm, the term **key** refers to a different mathematical property. For example:

- For RSA, the key is the product of two large prime numbers, also called the **modulus**.
- For AES and Elliptic Curve Cryptography (ECC), the key is only a sequence of randomly generated bytes.
  - In some cases, AES keys are derived from a master key or a passphrase using a Key Derivation Function (KDF) like PBKDF2 (Password-Based Key Derivation Function 2)

If an application uses a key that is considered short and **insecure**, the encrypted data is exposed to attacks aimed at getting at the plaintext.

In general, it is best practice to expect a breach: that a user or organization with malicious intent will perform cryptographic attacks on this data after obtaining it by other means.

What is the potential impact?

After retrieving encrypted data and performing cryptographic attacks on it on a given timeframe, attackers can recover the plaintext that encryption was supposed to protect.

Depending on the recovered data, the impact may vary.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Additional attack surface

By modifying the plaintext of the encrypted message, an attacker may be able to trigger additional vulnerabilities in the code. An attacker can further exploit a system to obtain more information.
Encrypted values are often considered trustworthy because it would not be possible for a third party to modify them under normal circumstances.

Breach of confidentiality and privacy

When encrypted data contains personal or sensitive information, its retrieval by an attacker can lead to privacy violations, identity theft, financial loss, reputational damage, or unauthorized access to confidential systems.

In this scenario, the company, its employees, users, and partners could be seriously affected.

The impact is twofold, as data breaches and exposure of encrypted data can undermine trust in the organization, as customers, clients and stakeholders may lose confidence in the organization’s ability to protect their sensitive data.

Legal and compliance issues

In many industries and locations, there are legal and compliance requirements to protect sensitive data. If encrypted data is compromised and the plaintext can be recovered, companies face legal consequences, penalties, or violations of privacy laws.

introduction

This vulnerability exposes encrypted data to attacks whose goal is to recover the plaintext.

how_to_fix

The following code examples either explicitly or implicitly generate keys. Note that there are differences in the size of the keys depending on the algorithm.

Due to the mathematical properties of the algorithms, the security requirements for the key size vary depending on the algorithm.
For example, a 256-bit ECC key provides about the same level of security as a 3072-bit RSA key and a 128-bit symmetric key.

Noncompliant code example

Here is an example of a private key generation with RSA:

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', {
    modulusLength: 1024,  // Noncompliant
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

Here is an example of a key generation with the Digital Signature Algorithm (DSA):

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPairSync('dsa', {
    modulusLength: 1024,  // Noncompliant
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

Here is an example of an Elliptic Curve (EC) initialization. It implicitly generates a private key whose size is indicated in the elliptic curve name:

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPair('ec', {
    namedCurve: 'secp112r2', // Noncompliant
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

Compliant solution

Here is an example of a private key generation with RSA:

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', {
    modulusLength: 2048,
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

Here is an example of a key generation with the Digital Signature Algorithm (DSA):

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPairSync('dsa', {
    modulusLength: 2048,
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

Here is an example of an Elliptic Curve (EC) initialization. It implicitly generates a private key whose size is indicated in the elliptic curve name:

```
const crypto = require('crypto');

function callback(err, pub, priv) {}

var { privateKey, publicKey } = crypto.generateKeyPair('ec', {
    namedCurve: 'secp224k1',
    publicKeyEncoding:  { type: 'spki', format: 'pem' },
    privateKeyEncoding: { type: 'pkcs8', format: 'pem' }
  },
  callback);
```

How does this work?

As a rule of thumb, use the cryptographic algorithms and mechanisms that are considered strong by the cryptography community.

The appropriate choices are the following.

RSA (Rivest-Shamir-Adleman) and DSA (Digital Signature Algorithm)

The security of these algorithms depends on the difficulty of attacks attempting to solve their underlying mathematical problem.

In general, a minimum key size of **2048** bits is recommended for both. It provides 112 bits of security. A key length of **3072** or **4096** should be preferred when possible.

AES (Advanced Encryption Standard)

AES supports three key sizes: 128 bits, 192 bits and 256 bits. The security of the AES algorithm is based on the computational complexity of trying all possible keys.
A larger key size increases the number of possible keys and makes exhaustive search attacks computationally infeasible. Therefore, a 256-bit key provides a higher level of security than a 128-bit or 192-bit key.

Currently, a minimum key size of **128 bits** is recommended for AES.

Elliptic Curve Cryptography (ECC)

Elliptic curve cryptography is also used in various algorithms, such as ECDSA, ECDH, or ECMQV. The length of keys generated with elliptic curve algorithms is mentioned directly in their names. For example, `secp256k1` generates a 256-bits long private key.

Currently, a minimum key size of **224 bits** is recommended for EC-based algorithms.

Additionally, some curves that theoretically provide sufficiently long keys are still discouraged. This can be because of a flaw in the curve parameters, a bad overall design, or poor performance. It is generally advised to use a NIST-approved elliptic curve wherever possible. Such curves currently include:

- NIST P curves with a size of at least 224 bits, e.g. secp256r1.
- Curve25519, generally known as ed25519 or x25519 depending on its application.
- Curve448.
- Brainpool curves with a size of at least 224 bits, e.g. brainpoolP224r1

Going the extra mile

Pre-Quantum Cryptography

Encrypted data and communications recorded today could be decrypted in the future by an attack from a quantum computer.
It is important to keep in mind that NIST-approved digital signature schemes, key agreement, and key transport may need to be replaced with secure quantum-resistant (or "post-quantum") counterpart.

Thus, if data is to remain secure beyond 2030, proactive measures should be taken now to ensure its safety.

Learn more here.

resources

- Documentation

  - NIST Documentation - NIST SP 800-186: Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters
  - IETF - rfc5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation

Articles & blog posts

- Microsoft, Timing vulnerabilities with CBC-mode symmetric decryption using padding
- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- NIST 800-131A - Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms

---

Disabling CSRF protections is security-sensitive

Clave: javascript:S4502

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn’t intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

default

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn’t intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

Ask Yourself Whether

- The web application uses cookies to authenticate users.
- There exist sensitive operations in the web application that can be performed when the user is authenticated.
- The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Protection against CSRF attacks is strongly recommended:
  - to be activated by default for all unsafe HTTP methods.
  - implemented, for example, with an unguessable CSRF token
- Of course all sensitive operations should not be performed with safe HTTP methods like `GET` which are designed to be used only for information retrieval.

Sensitive Code Example

Express.js CSURF middleware protection is not found on an unsafe HTTP method like POST method:

```
let csrf = require('csurf');
let express = require('express');

let csrfProtection = csrf({ cookie: true });
let app = express();

// Sensitive: this operation doesn't look like protected by CSURF middleware (csrfProtection is not used)
app.post('/money_transfer', parseForm, function (req, res) {
  res.send('Money transferred');
});
```

Protection provided by Express.js CSURF middleware is globally disabled on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

app.use(csrf({ cookie: true, ignoreMethods: ["POST", "GET"] })); // Sensitive as POST is unsafe method
```

Compliant Solution

Express.js CSURF middleware protection is used on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

let csrfProtection = csrf({ cookie:  true });

let app = express();

app.post('/money_transfer', parseForm, csrfProtection, function (req, res) { // Compliant
  res.send('Money transferred')
});
```

Protection provided by Express.js CSURF middleware is enabled on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

app.use(csrf({ cookie: true, ignoreMethods: ["GET"] })); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Cross-Site Request Forgery
- STIG Viewer - Application Security and Development: V-222603 - The application must protect from Cross-Site Request Forgery (CSRF) vulnerabilities.
- PortSwigger - Web storage: the lesser evil for session tokens

assess_the_problem

Ask Yourself Whether

- The web application uses cookies to authenticate users.
- There exist sensitive operations in the web application that can be performed when the user is authenticated.
- The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Express.js CSURF middleware protection is not found on an unsafe HTTP method like POST method:

```
let csrf = require('csurf');
let express = require('express');

let csrfProtection = csrf({ cookie: true });

let app = express();

// Sensitive: this operation doesn't look like protected by CSURF middleware (csrfProtection is not used)
app.post('/money_transfer', parseForm, function (req, res) {
  res.send('Money transferred');
});
```

Protection provided by Express.js CSURF middleware is globally disabled on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

app.use(csrf({ cookie: true, ignoreMethods: ["POST", "GET"] })); // Sensitive as POST is unsafe method
```

how_to_fix

Recommended Secure Coding Practices

- Protection against CSRF attacks is strongly recommended:
  - to be activated by default for all unsafe HTTP methods.
  - implemented, for example, with an unguessable CSRF token
- Of course all sensitive operations should not be performed with safe HTTP methods like `GET` which are designed to be used only for information retrieval.

Compliant Solution

Express.js CSURF middleware protection is used on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

let csrfProtection = csrf({ cookie:  true });

let app = express();

app.post('/money_transfer', parseForm, csrfProtection, function (req, res) { // Compliant
  res.send('Money transferred')
});
```

Protection provided by Express.js CSURF middleware is enabled on unsafe methods:

```
let csrf = require('csurf');
let express = require('express');

app.use(csrf({ cookie: true, ignoreMethods: ["GET"] })); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Cross-Site Request Forgery
- STIG Viewer - Application Security and Development: V-222603 - The application must protect from Cross-Site Request Forgery (CSRF) vulnerabilities.
- PortSwigger - Web storage: the lesser evil for session tokens

---

Delivering code in production with debug features activated is security-sensitive

Clave: javascript:S4507

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

default

Development tools and frameworks usually have options to make debugging easier for developers. Although these features are useful during development, they should never be enabled for applications deployed in production. Debug instructions or error messages can leak detailed information about the system, like the application’s path or file names.

Ask Yourself Whether

- The code or configuration enabling the application debug features is deployed on production servers or distributed to end users.
- The application runs by default with debug features activated.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Do not enable debugging features on production servers or applications distributed to end users.

Sensitive Code Example

errorhandler Express.js middleware should not be used in production:

```
const express = require('express');
const errorhandler = require('errorhandler');

let app = express();
app.use(errorhandler()); // Sensitive
```

Compliant Solution

errorhandler Express.js middleware used only in development mode:

```
const express = require('express');
const errorhandler = require('errorhandler');

let app = express();

if (process.env.NODE_ENV === 'development') {
  app.use(errorhandler());
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code
- CWE - CWE-215 - Information Exposure Through Debug Information

assess_the_problem

Ask Yourself Whether

- The code or configuration enabling the application debug features is deployed on production servers or distributed to end users.
- The application runs by default with debug features activated.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

errorhandler Express.js middleware should not be used in production:

```
const express = require('express');
const errorhandler = require('errorhandler');

let app = express();
app.use(errorhandler()); // Sensitive
```

root_cause

Development tools and frameworks usually have options to make debugging easier for developers. Although these features are useful during development, they should never be enabled for applications deployed in production. Debug instructions or error messages can leak detailed information about the system, like the application’s path or file names.

how_to_fix

Recommended Secure Coding Practices

Do not enable debugging features on production servers or applications distributed to end users.

Compliant Solution

errorhandler Express.js middleware used only in development mode:

```
const express = require('express');
const errorhandler = require('errorhandler');

let app = express();

if (process.env.NODE_ENV === 'development') {
  app.use(errorhandler());
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code
- CWE - CWE-215 - Information Exposure Through Debug Information

---

Template literals should not be nested

Clave: javascript:S4624

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Template literals (Template strings)

root_cause

Template literals, also known as template strings, allow for string interpolation and multiline strings in JavaScript. They provide a more convenient and flexible way to work with strings compared to traditional string concatenation or manipulation.

Template literals are delimited with the backtick (` ) character. They are a convenient way to include variables or expressions within a string using placeholders `${expression}` in the string and evaluate them dynamically.

However, nesting template literals can make the code less readable. With each nested template literal, the code becomes more complex and harder to understand. It can be challenging to keep track of the opening and closing backticks and properly escape characters if needed.

```
const color = "red";
const count = 3;
const message = `I have ${color ? `${count} ${color}` : count} apples`; // Noncompliant: nested template strings not easy to read
```

In such situations, moving the nested template into a separate statement is preferable.

```
const color = "red";
const count = 3;
const apples = color ? `${count} ${color}` : count;
const message = `I have ${apples} apples`;
```

The rule makes an exception for nested template literals spanning multiple lines. It allows you to visually separate different sections and gives you more freedom in formatting your text, including line breaks, indentation, and other formatting elements, enhancing readability.

```
const name = 'John';
const age = 42;

const message = `Hello ${name}!
You are ${age} years old.
${`This is a nested template literal.`}
It can span multiple lines.`;
```

Shorthand promises should be used

Clave: javascript:S4634

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Asynchronous JavaScript
- MDN web docs - Promise
- MDN web docs - Promise.prototype.then()
- MDN web docs - Promise.resolve()
- MDN web docs - Promise.reject()

root_cause

In JavaScript, a promise is an object representing the eventual completion or failure of an asynchronous operation. It is a way to handle asynchronous operations more elegantly and avoid the "callback hell".

A promise can be in one of three states:

- Pending: The initial state of a promise. It represents that the asynchronous operation is still ongoing and has not yet been fulfilled or rejected.
- Fulfilled: The state of a promise when the asynchronous operation has been successfully completed. It represents that the promised value is available and can be consumed.
- Rejected: The state of a promise when the asynchronous operation encounters an error or fails to complete. It represents that an error has occurred and the promised value is not available.

The basic syntax for creating a promise in JavaScript is as follows:

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  // If the operation is successful, call resolve(value)
  // If the operation fails, call reject(error)
});
```

However, when it comes to immediately resolving or rejecting states, creating a new promise with the `Promise` constructor and manually calling `resolve` or `reject` makes the code verbose and more difficult to read.

```
const result = new Promise(resolve => resolve(42)); // Noncompliant: Redundant to explicitly create a promise to resolve 42
result.then(value => {
  console.log(value); // Output: 42
});
```

Instead, a promise can be created with `Promise.resolve`. It is typically used when you want to create a new promise that is already resolved with a certain value. It is commonly used to wrap synchronous values or functions into promises.

how_to_fix

If you already have a synchronous value that you want to convert into a promise, using `Promise.resolve` is more concise and straightforward. It immediately creates a promise that is already resolved with the provided value.

```
const result = Promise.resolve(42);
result.then(value => {
  console.log(value); // Output: 42
});
```

Similarly, if you have an error or an exceptional condition and want to create a promise that is immediately rejected with that error, using `Promise.reject` is more straightforward. It creates a promise in the rejected state with the provided error.

```
const error = new Error('Something went wrong');
const promise = Promise.reject(error);
```

If you have a condition and want to create a promise that is either resolved or rejected based on that condition, using `Promise.resolve` or `Promise.reject` helps make the code more readable and concise.

```
function fetchData() {
  if (cache) {
    return Promise.resolve(cache);
  } else if (shouldFetchData()) {
    return fetchDataFromServer()
        .then(data => {
            cache = data;
            return data;
        });
  } else {
    return Promise.reject(new Error('Data fetch is not required'));
  }
}
```

When you have a promise chain and want to introduce an intermediate step with an immediately resolved value, using `Promise.resolve` allows you to continue the chain without introducing unnecessary complexity.

```
const data = cache ? cache : fetchData();

return Promise.resolve(data) // data may be a Promise or not, we need to wrap it
    .then(data => {
        return sanitizeData(data);
    })
```

Using `Promise.resolve` and `Promise.reject` is particularly useful when you want to simplify the creation of promises with immediately resolved or rejected states. They provide a cleaner and more direct approach compared to creating a new promise with the `Promise` constructor and manually calling `resolve` or `reject`.

Encrypting data is security-sensitive

Clave: javascript:S4787

Severidad: CRITICAL

Estado: ⚠ Depreciada

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

- Generate encryption keys using secure random algorithms.
- When generating cryptographic keys (or key pairs), it is important to use a key length that provides enough entropy against brute-force attacks. For the Blowfish algorithm the key should be at least 128 bits long, while for the RSA algorithm it should be at least 2048 bits long.
- Regenerate the keys regularly.
- Always store the keys in a safe location and transfer them only over safe channels.
- If there is an exchange of cryptographic keys, check first the identity of the receiver.

- Only use strong encryption algorithms. Check regularly that the algorithm is still deemed secure. It is also imperative that they are implemented correctly. Use only encryption libraries which are deemed secure. Do not define your own encryption algorithms as they will most probably have flaws.
- When a nonce is used, generate it randomly every time.
- When using the RSA algorithm, incorporate an Optimal Asymmetric Encryption Padding (OAEP).
- When CBC is used for encryption, the IV must be random and unpredictable. Otherwise it exposes the encrypted value to crypto-analysis attacks like "Chosen-Plaintext Attacks". Thus a secure random algorithm should be used. An IV value should be associated to one and only one encryption cycle, because the IV’s purpose is to ensure that the same plaintext encrypted twice will yield two different ciphertexts.
- The Advanced Encryption Standard (AES) encryption algorithm can be used with various modes. Galois/Counter Mode (GCM) with no padding should be preferred to the following combinations which are not secured:
  - Electronic Codebook (ECB) mode: Under a given key, any given plaintext block always gets encrypted to the same ciphertext block. Thus, it does not hide data patterns well. In some senses, it doesn’t provide serious message confidentiality, and it is not recommended for use in cryptographic protocols at all.
  - Cipher Block Chaining (CBC) with PKCS#5 padding (or PKCS#7) is susceptible to padding oracle attacks.

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-321 - Use of Hard-coded Cryptographic Key
- CWE - CWE-322 - Key Exchange without Entity Authentication
- CWE - CWE-323 - Reusing a Nonce, Key Pair in Encryption
- CWE - CWE-324 - Use of a Key Past its Expiration Date
- CWE - CWE-325 - Missing Required Cryptographic Step
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm

default

This rule is deprecated; use S4426, S5542, S5547 instead.

Encrypting data is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-7902
- CVE-2006-1378
- CVE-2003-1376

Proper encryption requires both the encryption algorithm and the key to be strong. Obviously the private key needs to remain secret and be renewed regularly. However these are not the only means to defeat or weaken an encryption.

This rule flags function calls that initiate encryption/decryption.

Ask Yourself Whether

- the private key might not be random, strong enough or the same key is reused for a long long time.
- the private key might be compromised. It can happen when it is stored in an unsafe place or when it was transferred in an unsafe manner.
- the key exchange is made without properly authenticating the receiver.
- the encryption algorithm is not strong enough for the level of protection required. Note that encryption algorithms strength decreases as time passes.
- the chosen encryption library is deemed unsafe.
- a nonce is used, and the same value is reused multiple times, or the nonce is not random.
- the RSA algorithm is used, and it does not incorporate an Optimal Asymmetric Encryption Padding (OAEP), which might weaken the encryption.
- the CBC (Cypher Block Chaining) algorithm is used for encryption, and it’s IV (Initialization Vector) is not generated using a secure random algorithm, or it is reused.
- the Advanced Encryption Standard (AES) encryption algorithm is used with an unsecure mode. See the recommended practices for more information.

You are at risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Generate encryption keys using secure random algorithms.
- When generating cryptographic keys (or key pairs), it is important to use a key length that provides enough entropy against brute-force attacks. For the Blowfish algorithm the key should be at least 128 bits long, while for the RSA algorithm it should be at least 2048 bits long.
- Regenerate the keys regularly.
- Always store the keys in a safe location and transfer them only over safe channels.
- If there is an exchange of cryptographic keys, check first the identity of the receiver.
- Only use strong encryption algorithms. Check regularly that the algorithm is still deemed secure. It is also imperative that they are implemented correctly. Use only encryption libraries which are deemed secure. Do not define your own encryption algorithms as they will most probably have flaws.
- When a nonce is used, generate it randomly every time.
- When using the RSA algorithm, incorporate an Optimal Asymmetric Encryption Padding (OAEP).
- When CBC is used for encryption, the IV must be random and unpredictable. Otherwise it exposes the encrypted value to crypto-analysis attacks like "Chosen-Plaintext Attacks". Thus a secure random algorithm should be used. An IV value should be associated to one and only one encryption cycle, because the IV’s purpose is to ensure that the same plaintext encrypted twice will yield two different ciphertexts.

- The Advanced Encryption Standard (AES) encryption algorithm can be used with various modes. Galois/Counter Mode (GCM) with no padding should be preferred to the following combinations which are not secured:
    - Electronic Codebook (ECB) mode: Under a given key, any given plaintext block always gets encrypted to the same ciphertext block. Thus, it does not hide data patterns well. In some senses, it doesn't provide serious message confidentiality, and it is not recommended for use in cryptographic protocols at all.
    - Cipher Block Chaining (CBC) with PKCS#5 padding (or PKCS#7) is susceptible to padding oracle attacks.

Sensitive Code Example

```
// === Client side ===
crypto.subtle.encrypt(algo, key, plainData); // Sensitive
crypto.subtle.decrypt(algo, key, encData); // Sensitive

// === Server side ===
const crypto = require("crypto");
const cipher = crypto.createCipher(algo, key); // Sensitive
const cipheriv = crypto.createCipheriv(algo, key, iv); // Sensitive
const decipher = crypto.createDecipher(algo, key); // Sensitive
const decipheriv = crypto.createDecipheriv(algo, key, iv); // Sensitive
const pubEnc = crypto.publicEncrypt(key, buf); // Sensitive
const privDec = crypto.privateDecrypt({ key: key, passphrase: secret }, pubEnc); // Sensitive
const privEnc = crypto.privateEncrypt({ key: key, passphrase: secret }, buf); // Sensitive
const pubDec = crypto.publicDecrypt(key, privEnc); // Sensitive
```

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-321 - Use of Hard-coded Cryptographic Key
- CWE - CWE-322 - Key Exchange without Entity Authentication
- CWE - CWE-323 - Reusing a Nonce, Key Pair in Encryption
- CWE - CWE-324 - Use of a Key Past its Expiration Date
- CWE - CWE-325 - Missing Required Cryptographic Step
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm

root_cause

This rule is deprecated; use S4426, S5542, S5547 instead.

Encrypting data is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-7902
- CVE-2006-1378
- CVE-2003-1376

Proper encryption requires both the encryption algorithm and the key to be strong. Obviously the private key needs to remain secret and be renewed regularly. However these are not the only means to defeat or weaken an encryption.

This rule flags function calls that initiate encryption/decryption.

assess_the_problem

Ask Yourself Whether

- the private key might not be random, strong enough or the same key is reused for a long long time.
- the private key might be compromised. It can happen when it is stored in an unsafe place or when it was transferred in an unsafe manner.
- the key exchange is made without properly authenticating the receiver.
- the encryption algorithm is not strong enough for the level of protection required. Note that encryption algorithms strength decreases as time passes.
- the chosen encryption library is deemed unsafe.
- a nonce is used, and the same value is reused multiple times, or the nonce is not random.
- the RSA algorithm is used, and it does not incorporate an Optimal Asymmetric Encryption Padding (OAEP), which might weaken the encryption.
- the CBC (Cypher Block Chaining) algorithm is used for encryption, and it's IV (Initialization Vector) is not generated using a secure random algorithm, or it is reused.
- the Advanced Encryption Standard (AES) encryption algorithm is used with an unsecure mode. See the recommended practices for more information.

You are at risk if you answered yes to any of those questions.

Sensitive Code Example

```
// === Client side ===
crypto.subtle.encrypt(algo, key, plainData); // Sensitive
crypto.subtle.decrypt(algo, key, encData); // Sensitive

// === Server side ===
const crypto = require("crypto");
const cipher = crypto.createCipher(algo, key); // Sensitive
const cipheriv = crypto.createCipheriv(algo, key, iv); // Sensitive
const decipher = crypto.createDecipher(algo, key); // Sensitive
const decipheriv = crypto.createDecipheriv(algo, key, iv); // Sensitive
const pubEnc = crypto.publicEncrypt(key, buf); // Sensitive
const privDec = crypto.privateDecrypt({ key: key, passphrase: secret }, pubEnc); // Sensitive
const privEnc = crypto.privateEncrypt({ key: key, passphrase: secret }, buf); // Sensitive
const pubDec = crypto.publicDecrypt(key, privEnc); // Sensitive
```

Encryption algorithms should be used with secure mode and padding scheme

Clave: javascript:S5542

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

introduction

This vulnerability exposes encrypted data to a number of attacks whose goal is to recover the plaintext.

how_to_fix

Noncompliant code example

Example with a symmetric cipher, AES:

```
const crypto = require('crypto');
```

```
crypto.createCipheriv("AES-128-CBC", key, iv); // Noncompliant
```

Compliant solution

Example with a symmetric cipher, AES:

```
const crypto = require('crypto');
```

```
crypto.createCipheriv("AES-256-GCM", key, iv);
```

How does this work?

As a rule of thumb, use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

Appropriate choices are currently the following.

For AES: use authenticated encryption modes

The best-known authenticated encryption mode for AES is Galois/Counter mode (GCM).

GCM mode combines encryption with authentication and integrity checks using a cryptographic hash function and provides both confidentiality and authenticity of data.

Other similar modes are:

- CCM: `Counter with CBC-MAC`
- CWC: `Cipher Block Chaining with Message Authentication Code`
- EAX: `Encrypt-and-Authenticate`
- IAPM: `Integer Authenticated Parallelizable Mode`
- OCB: `Offset Codebook Mode`

It is also possible to use AES-CBC with HMAC for integrity checks. However, it is considered more straightforward to use AES-GCM directly instead.

For RSA: use the OAEP scheme

The Optimal Asymmetric Encryption Padding scheme (OAEP) adds randomness and a secure hash function that strengthens the regular inner workings of RSA.

root_cause

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

For AES, the weakest mode is ECB (Electronic Codebook). Repeated blocks of data are encrypted to the same value, making them easy to identify and reducing the difficulty of recovering the original cleartext.

Unauthenticated modes such as CBC (Cipher Block Chaining) may be used but are prone to attacks that manipulate the ciphertext. They must be used with caution.

For RSA, the weakest algorithms are either using it without padding or using the PKCS1v1.5 padding scheme.

What is the potential impact?

The cleartext of an encrypted message might be recoverable. Additionally, it might be possible to modify the cleartext of an encrypted message.

Below are some real-world scenarios that illustrate possible impacts of an attacker exploiting the vulnerability.

Theft of sensitive data

The encrypted message might contain data that is considered sensitive and should not be known to third parties.

By using a weak algorithm the likelihood that an attacker might be able to recover the cleartext drastically increases.

Additional attack surface

By modifying the cleartext of the encrypted message it might be possible for an attacker to trigger other vulnerabilities in the code. Encrypted values are often considered trusted, since under normal circumstances it would not be possible for a third party to modify them.

resources

Articles & blog posts

- Microsoft, Timing vulnerabilities with CBC-mode symmetric decryption using padding
- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm

---

Cipher algorithms should be robust

Clave: javascript:S5547

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Encryption algorithms are essential for protecting sensitive information and ensuring secure communication in various domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

What is the potential impact?

The cleartext of an encrypted message might be recoverable. Additionally, it might be possible to modify the cleartext of an encrypted message.

Below are some real-world scenarios that illustrate some impacts of an attacker exploiting the vulnerability.

Theft of sensitive data

The encrypted message might contain data that is considered sensitive and should not be known to third parties.

By using a weak algorithm the likelihood that an attacker might be able to recover the cleartext drastically increases.

Additional attack surface

By modifying the cleartext of the encrypted message it might be possible for an attacker to trigger other vulnerabilities in the code. Encrypted values are often considered trusted, since under normal circumstances it would not be possible for a third party to modify them.

resources

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- STIG Viewer - Application Security and Development: V-222396 - The application must implement DoD-approved encryption to protect the confidentiality of remote access sessions.

how_to_fix

The following code contains examples of algorithms that are not considered highly resistant to cryptanalysis and thus should be avoided.

Noncompliant code example

```
const crypto = require('crypto');

crypto.createCipheriv("DES", key, iv); // Noncompliant
```

Compliant solution

```
const crypto = require('crypto');

crypto.createCipheriv("AES-256-GCM", key, iv);
```

How does this work?

Use a secure algorithm

It is highly recommended to use an algorithm that is currently considered secure by the cryptographic community. A common choice for such an algorithm is the Advanced Encryption Standard (AES).

For block ciphers, it is not recommended to use algorithms with a block size that is smaller than 128 bits.

introduction

This vulnerability makes it possible that the cleartext of the encrypted message might be recoverable without prior knowledge of the key.

Using intrusive permissions is security-sensitive

Clave: javascript:S5604

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Powerful features are browser features (geolocation, camera, microphone …) that can be accessed with JavaScript API and may require a permission granted by the user. These features can have a high impact on privacy and user security thus they should only be used if they are really necessary to implement the critical parts of an application.

This rule highlights intrusive permissions when requested with the future standard (but currently experimental) web browser query API and specific APIs related to the permission. It is highly recommended to customize this rule with the permissions considered as intrusive in the context of the web application.

Ask Yourself Whether

- Some powerful features used by the application are not really necessary.
- Users are not clearly informed why and when powerful features are used by the application.

You are at risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- In order to respect user privacy it is recommended to avoid using intrusive powerful features.

Sensitive Code Example

When using geolocation API, Firefox for example retrieves personal information like nearby wireless access points and IP address and sends it to the default geolocation service provider, Google Location Services:

```
navigator.permissions.query({name:"geolocation"}).then(function(result) {
});  // Sensitive: geolocation is a powerful feature with high privacy concerns

navigator.geolocation.getCurrentPosition(function(position) {
  console.log("coordinates x="+position.coords.latitude+" and y="+position.coords.longitude);
}); // Sensitive: geolocation is a powerful feature with high privacy concerns
```

Compliant Solution

If geolocation is required, always explain to the user why the application needs it and prefer requesting an approximate location when possible:

```
<html>
<head>
    <title>
        Retailer website example
    </title>
</head>
<body>
    Type a city, street or zip code where you want to retrieve the closest retail locations of our products:
    <form method=post>
        <input type=text value="New York"> <!-- Compliant -->
    </form>
</body>
</html>
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Web Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-250 - Execution with Unnecessary Privileges
- CWE - CWE-359 - Exposure of Private Information
- W3C - Permissions
- Mozilla - Does Firefox share my location with websites?

how_to_fix

Recommended Secure Coding Practices

- In order to respect user privacy it is recommended to avoid using intrusive powerful features.

Compliant Solution

If geolocation is required, always explain to the user why the application needs it and prefer requesting an approximate location when possible:

```
<html>
<head>
    <title>
        Retailer website example
    </title>
</head>
<body>
    Type a city, street or zip code where you want to retrieve the closest retail locations of our products:
    <form method=post>
        <input type=text value="New York"> <!-- Compliant -->
    </form>
```

```
</body>
</html>
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Web Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-250 - Execution with Unnecessary Privileges
- CWE - CWE-359 - Exposure of Private Information
- W3C - Permissions
- Mozilla - Does Firefox share my location with websites?

assess_the_problem

Ask Yourself Whether

- Some powerful features used by the application are not really necessary.
- Users are not clearly informed why and when powerful features are used by the application.

You are at risk if you answered yes to any of those questions.

Sensitive Code Example

When using geolocation API, Firefox for example retrieves personal information like nearby wireless access points and IP address and sends it to the default geolocation service provider, Google Location Services:

```
navigator.permissions.query({name:"geolocation"}).then(function(result) {
});  // Sensitive: geolocation is a powerful feature with high privacy concerns

navigator.geolocation.getCurrentPosition(function(position) {
  console.log("coordinates x="+position.coords.latitude+" and y="+position.coords.longitude);
}); // Sensitive: geolocation is a powerful feature with high privacy concerns
```

root_cause

Powerful features are browser features (geolocation, camera, microphone …) that can be accessed with JavaScript API and may require a permission granted by the user. These features can have a high impact on privacy and user security thus they should only be used if they are really necessary to implement the critical parts of an application.

This rule highlights intrusive permissions when requested with the future standard (but currently experimental) web browser query API and specific APIs related to the permission. It is highly recommended to customize this rule with the permissions considered as intrusive in the context of the web application.

JWT should be signed and verified with strong cipher algorithms

Clave: javascript:S5659

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

introduction

This vulnerability allows forging of JSON Web Tokens to impersonate other users.

root_cause

JSON Web Tokens (JWTs), a popular method of securely transmitting information between parties as a JSON object, can become a significant security risk when they are not properly signed with a robust cipher algorithm, left unsigned altogether, or if the signature is not verified. This vulnerability class allows malicious actors to craft fraudulent tokens, effectively impersonating user identities. In essence, the integrity of a JWT hinges on the strength and presence of its signature.

What is the potential impact?

When a JSON Web Token is not appropriately signed with a strong cipher algorithm or if the signature is not verified, it becomes a significant threat to data security and the privacy of user identities.

Impersonation of users

JWTs are commonly used to represent user authorization claims. They contain information about the user’s identity, user roles, and access rights. When these tokens are not securely signed, it allows an attacker to forge them. In essence, a weak or missing signature gives an attacker the power to craft a token that could impersonate any user. For instance, they could create a token for an administrator account, gaining access to high-level permissions and sensitive data.

Unauthorized data access

When a JWT is not securely signed, it can be tampered with by an attacker, and the integrity of the data it carries cannot be trusted. An attacker can manipulate the content of the token and grant themselves permissions they should not have, leading to unauthorized data access.

how_to_fix

The following code contains examples of JWT encoding and decoding without a strong cipher algorithm.

Noncompliant code example

```
const jwt = require('jsonwebtoken');

jwt.sign(payload, key, { algorithm: 'none' }); // Noncompliant
```

```
const jwt = require('jsonwebtoken');

jwt.verify(token, key, {
    expiresIn: 360000,
    algorithms: ['none'] // Noncompliant
}, callbackcheck);
```

Compliant solution

```
const jwt = require('jsonwebtoken');

jwt.sign(payload, key, { algorithm: 'HS256' });
```

```
const jwt = require('jsonwebtoken');

jwt.verify(token, key, {
    expiresIn: 360000,
    algorithms: ['HS256']
}, callbackcheck);
```

How does this work?

Always sign your tokens

The foremost measure to enhance JWT security is to ensure that every JWT you issue is signed. Unsigned tokens are like open books that anyone can tamper with. Signing your JWTs ensures that any alterations to the tokens after they have been issued can be detected. Most JWT libraries support a signing function, and using it is usually as simple as providing a secret key when the token is created.

Choose a strong cipher algorithm

It is not enough to merely sign your tokens. You need to sign them with a strong cipher algorithm. Algorithms like HS256 (HMAC using SHA-256) are considered secure for most purposes. But for an additional layer of security, you could use an algorithm like RS256 (RSA Signature with SHA-256), which uses a private key for signing and a public key for verification. This way, even if someone gains access to the public key, they will not be able to forge tokens.

Verify the signature of your tokens

Resolving a vulnerability concerning the validation of JWT token signatures is mainly about incorporating a critical step into your process: validating the signature every time a token is decoded. Just having a signed token using a secure algorithm is not enough. If you are not validating signatures, they are not serving their purpose.

Every time your application receives a JWT, it needs to decode the token to extract the information contained within. It is during this decoding process that the signature of the JWT should also be checked.

To resolve the issue, follow these instructions:

1. Use framework-specific functions for signature verification: Most programming frameworks that support JWTs provide specific functions to not only decode a token but also validate its signature simultaneously. Make sure to use these functions when handling incoming tokens.
2. Handle invalid signatures appropriately: If a JWT’s signature does not validate correctly, it means the token is not trustworthy, indicating potential tampering. The action to take when encountering an invalid token should be denying the request carrying it and logging the event for further investigation.
3. Incorporate signature validation in your tests: When you are writing tests for your application, include tests that check the signature validation functionality. This can help you catch any instances where signature verification might be unintentionally skipped or bypassed.

By following these practices, you can ensure the security of your application’s JWT handling process, making it resistant to attacks that rely on tampering with tokens. Validation of the signature needs to be an integral and non-negotiable part of your token handling process.

Going the extra mile

Securely store your secret keys

Ensure that your secret keys are stored securely. They should not be hard-coded into your application code or checked into your version control system. Instead, consider using environment variables, secure key management systems, or vault services.

Rotate your secret keys

Even with the strongest cipher algorithms, there is a risk that your secret keys may be compromised. Therefore, it is a good practice to periodically rotate your secret keys. By doing so, you limit the amount of time that an attacker can misuse a stolen key. When you rotate keys, be sure to allow a grace period where tokens signed with the old key are still accepted to prevent service disruptions.

resources

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-347 - Improper Verification of Cryptographic Signature

---

Using remote artifacts without integrity checks is security-sensitive

Clave: javascript:S5725

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using remote artifacts without integrity checks can lead to the unexpected execution of malicious code in the application.

On the client side, where front-end code is executed, malicious code could:

- impersonate users' identities and take advantage of their privileges on the application.
- add quiet malware that monitors users' session and capture sensitive secrets.
- gain access to sensitive clients' personal data.
- deface, or otherwise affect the general availability of the application.
- mine cryptocurrencies in the background.

Likewise, a compromised software piece that would be deployed on a server-side application could badly affect the application’s security. For example, server-side malware could:

- access and modify sensitive technical and business data.
- elevate its privileges on the underlying operating system.
- Use the compromised application as a pivot to attack the local network.

By ensuring that a remote artifact is exactly what it is supposed to be before using it, the application is protected from unexpected changes applied to it before it is downloaded.
Especially, integrity checks will allow for identifying an artifact replaced by malware on the publication website or that was legitimately changed by its author, in a more benign scenario.

Important note: downloading an artifact over HTTPS only protects it while in transit from one host to another. It provides authenticity and integrity checks **for the network stream** only. It does not ensure the authenticity or security of the artifact itself.

assess_the_problem

Ask Yourself Whether

- The artifact is a file intended to execute code.
- The artifact is a file that is intended to configure or affect running code in some way.

There is a risk if you answered yes to any of these questions.

Sensitive Code Example

The following code sample uses neither integrity checks nor version pinning:

```
let script = document.createElement("script");
script.src = "https://cdn.example.com/latest/script.js"; // Sensitive
script.crossOrigin = "anonymous";
document.head.appendChild(script);
```

default

Using remote artifacts without integrity checks can lead to the unexpected execution of malicious code in the application.

On the client side, where front-end code is executed, malicious code could:

- impersonate users' identities and take advantage of their privileges on the application.
- add quiet malware that monitors users' session and capture sensitive secrets.
- gain access to sensitive clients' personal data.
- deface, or otherwise affect the general availability of the application.
- mine cryptocurrencies in the background.

Likewise, a compromised software piece that would be deployed on a server-side application could badly affect the application's security. For example, server-side malware could:

- access and modify sensitive technical and business data.
- elevate its privileges on the underlying operating system.
- Use the compromised application as a pivot to attack the local network.

By ensuring that a remote artifact is exactly what it is supposed to be before using it, the application is protected from unexpected changes applied to it before it is downloaded.
Especially, integrity checks will allow for identifying an artifact replaced by malware on the publication website or that was legitimately changed by its author, in a more benign scenario.

Important note: downloading an artifact over HTTPS only protects it while in transit from one host to another. It provides authenticity and integrity checks **for the network stream** only. It does not ensure the authenticity or security of the artifact itself.

Ask Yourself Whether

- The artifact is a file intended to execute code.
- The artifact is a file that is intended to configure or affect running code in some way.

There is a risk if you answered yes to any of these questions.

Recommended Secure Coding Practices

To check the integrity of a remote artifact, hash verification is the most reliable solution. It does ensure that the file has not been modified since the fingerprint was computed.

In this case, the artifact's hash must:

- Be computed with a secure hash algorithm such as `SHA512`, `SHA384` or `SHA256`.
- Be compared with a secure hash that was **not** downloaded from the same source.

To do so, the best option is to add the hash in the code explicitly, by following Mozilla's official documentation on how to generate integrity strings.

Note: Use this fix together with version binding on the remote file. Avoid downloading files named "latest" or similar, so that the front-end pages do not break when the code of the latest remote artifact changes.

Sensitive Code Example

The following code sample uses neither integrity checks nor version pinning:

```
let script = document.createElement("script");
script.src = "https://cdn.example.com/latest/script.js"; // Sensitive
script.crossOrigin = "anonymous";
document.head.appendChild(script);
```

Compliant Solution

```
let script = document.createElement("script");
script.src = "https://cdn.example.com/v5.3.6/script.js";
script.integrity = "sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC";
script.crossOrigin = "anonymous";
document.head.appendChild(script);
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- CWE - CWE-353 - Missing Support for Integrity Check
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - Subresource Integrity
- Wikipedia, Watering Hole Attacks

how_to_fix

Recommended Secure Coding Practices

To check the integrity of a remote artifact, hash verification is the most reliable solution. It does ensure that the file has not been modified since the fingerprint was computed.

In this case, the artifact’s hash must:

- Be computed with a secure hash algorithm such as `SHA512`, `SHA384` or `SHA256`.
- Be compared with a secure hash that was **not** downloaded from the same source.

To do so, the best option is to add the hash in the code explicitly, by following Mozilla’s official documentation on how to generate integrity strings.

Note: Use this fix together with version binding on the remote file. Avoid downloading files named "latest" or similar, so that the front-end pages do not break when the code of the latest remote artifact changes.

Compliant Solution

```
let script = document.createElement("script");
script.src = "https://cdn.example.com/v5.3.6/script.js";
script.integrity = "sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC";
script.crossOrigin = "anonymous";
document.head.appendChild(script);
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- CWE - CWE-353 - Missing Support for Integrity Check
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - Subresource Integrity
- Wikipedia, Watering Hole Attacks

Disabling content security policy fetch directives is security-sensitive

Clave: javascript:S5728

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

Implement content security policy fetch directives, in particular *default-src* directive and continue to properly sanitize and validate all inputs of the application, indeed CSP fetch directives is only a tool to reduce the impact of cross site scripting attacks.

Compliant Solution

In a Express.js application, a standard way to implement CSP is the helmet contentSecurityPolicy middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();
app.use(helmet.contentSecurityPolicy()); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- w3.org - Content Security Policy Level 3
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - Content Security Policy (CSP)

root_cause

Content security policy (CSP) (fetch directives) is a W3C standard which is used by a server to specify, via a http header, the origins from where the browser is allowed to load resources. It can help to mitigate the risk of cross site scripting (XSS) attacks and reduce privileges used by an application. If the website doesn’t define CSP header the browser will apply same-origin policy by default.

```
Content-Security-Policy: default-src 'self'; script-src ‘self ‘ http://www.example.com
```

In the above example, all resources are allowed from the website where this header is set and script resources fetched from example.com are also authorized:

```
<img src="selfhostedimage.png></script> <!-- will be loaded because default-src 'self'; directive is applied  -->
<img src="http://www.example.com/image.png></script>  <!-- will NOT be loaded because default-src 'self'; directive is applied  -->
```

```
<script src="http://www.example.com/library.js></script> <!-- will be loaded because script-src ‘self ‘ http://www.example.comdirective is applied  -->
<script src="selfhostedscript.js></script> <!-- will be loaded because script-src ‘self ‘ http://www.example.com directive is applied  -->
<script src="http://www.otherexample.com/library.js></script> <!-- will NOT be loaded because script-src ‘self ‘ http://www.example.comdirective is appl
```

default

Content security policy (CSP) (fetch directives) is a W3C standard which is used by a server to specify, via a http header, the origins from where the browser is allowed to load resources. It can help to mitigate the risk of cross site scripting (XSS) attacks and reduce privileges used by an application. If the website doesn’t define CSP header the browser will apply same-origin policy by default.

```
Content-Security-Policy: default-src 'self'; script-src ‘self ‘ http://www.example.com
```

In the above example, all resources are allowed from the website where this header is set and script resources fetched from example.com are also authorized:

```
<img src="selfhostedimage.png></script> <!-- will be loaded because default-src 'self'; directive is applied  -->
<img src="http://www.example.com/image.png></script>  <!-- will NOT be loaded because default-src 'self'; directive is applied  -->
<script src="http://www.example.com/library.js></script> <!-- will be loaded because script-src ‘self ‘ http://www.example.comdirective is applied  -->
<script src="selfhostedscript.js></script> <!-- will be loaded because script-src ‘self ‘ http://www.example.com directive is applied  -->
<script src="http://www.otherexample.com/library.js></script> <!-- will NOT be loaded because script-src ‘self ‘ http://www.example.comdirective is appl
```

Ask Yourself Whether

- The resources of the application are fetched from various untrusted locations.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement content security policy fetch directives, in particular *default-src* directive and continue to properly sanitize and validate all inputs of the application, indeed CSP fetch directives is only a tool to reduce the impact of cross site scripting attacks.

Sensitive Code Example

In a Express.js application, the code is sensitive if the helmet contentSecurityPolicy middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();
app.use(
    helmet({
      contentSecurityPolicy: false, // sensitive
    })
);
```

Compliant Solution

In a Express.js application, a standard way to implement CSP is the helmet contentSecurityPolicy middleware:

```
const express = require('express');
const helmet = require('helmet');

let app = express();
app.use(helmet.contentSecurityPolicy()); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- w3.org - Content Security Policy Level 3
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- developer.mozilla.org - Content Security Policy (CSP)

assess_the_problem

Ask Yourself Whether

- The resources of the application are fetched from various untrusted locations.

There is a risk if you answered yes to this question.

Sensitive Code Example

In a Express.js application, the code is sensitive if the helmet contentSecurityPolicy middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();
```

```
app.use(
    helmet({
      contentSecurityPolicy: false, // sensitive
    })
);
```

Disabling Certificate Transparency monitoring is security-sensitive

Clave: javascript:S5742

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

default

This rule is deprecated, and will eventually be removed.

Certificate Transparency (CT) is an open-framework to protect against identity theft when certificates are issued. Certificate Authorities (CA) electronically sign certificate after verifying the identify of the certificate owner. Attackers use, among other things, social engineering attacks to trick a CA to correctly verifying a spoofed identity/forged certificate.

CAs implement Certificate Transparency framework to publicly log the records of newly issued certificates, allowing the public and in particular the identity owner to monitor these logs to verify that his identify was not usurped.

Ask Yourself Whether

- The website identity is valuable and well-known, therefore prone to theft.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Implement Expect-CT HTTP header which instructs the web browser to check public CT logs in order to verify if the website appears inside and if it is not, the browser will block the request and display a warning to the user.

Sensitive Code Example

In Express.js application the code is sensitive if the expect-ct middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
    helmet({
      expectCt: false // Sensitive
    })
);
```

Compliant Solution

In Express.js application the expect-ct middleware is the standard way to implement expect-ct. Usually, the deployment of this policy starts with the report only mode (`enforce: false`) and with a low `maxAge` (the number of seconds the policy will apply) value and next if everything works well it is recommended to block future connections that violate Expect-CT policy (`enforce: true`) and greater value for maxAge directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.expectCt({
  enforce: true,
  maxAge: 86400
})); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Certificate Transparency
- wikipedia.org - Certificate Authority

how_to_fix

Recommended Secure Coding Practices

Implement Expect-CT HTTP header which instructs the web browser to check public CT logs in order to verify if the website appears inside and if it is not, the browser will block the request and display a warning to the user.

Compliant Solution

In Express.js application the expect-ct middleware is the standard way to implement expect-ct. Usually, the deployment of this policy starts with the report only mode (`enforce: false`) and with a low `maxAge` (the number of seconds the policy will apply) value and next if everything works well it is recommended to block future connections that violate Expect-CT policy (`enforce: true`) and greater value for maxAge directive:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(helmet.expectCt({
  enforce: true,
  maxAge: 86400
})); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - Certificate Transparency
- wikipedia.org - Certificate Authority

assess_the_problem

Ask Yourself Whether

- The website identity is valuable and well-known, therefore prone to theft.

There is a risk if you answered yes to this question.

Sensitive Code Example

In Express.js application the code is sensitive if the expect-ct middleware is disabled:

```
const express = require('express');
const helmet = require('helmet');

let app = express();

app.use(
    helmet({
      expectCt: false // Sensitive
    })
);
```

root_cause

This rule is deprecated, and will eventually be removed.

Certificate Transparency (CT) is an open-framework to protect against identity theft when certificates are issued. Certificate Authorities (CA) electronically sign certificate after verifying the identify of the certificate owner. Attackers use, among other things, social engineering attacks to trick a CA to correctly verifying a spoofed identity/forged certificate.

CAs implement Certificate Transparency framework to publicly log the records of newly issued certificates, allowing the public and in particular the identity owner to monitor these logs to verify that his identify was not usurped.

Allowing confidential information to be logged is security-sensitive

Clave: javascript:S5757

Severidad: MINOR

Estado: ✓ Activa

Descripción: No disponible

default

Log management is an important topic, especially for the security of a web application, to ensure user activity, including potential attackers, is recorded and available for an analyst to understand what’s happened on the web application in case of malicious activities.

Retention of specific logs for a defined period of time is often necessary to comply with regulations such as GDPR, PCI DSS and others. However, to protect user’s privacy, certain informations are forbidden or strongly discouraged from being logged, such as user passwords or credit card numbers, which obviously should not be stored or at least not in clear text.

Ask Yourself Whether

In a production environment:

- The web application uses confidential information and logs a significant amount of data.
- Logs are externalized to SIEM or Big Data repositories.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Loggers should be configured with a list of confidential, personal information that will be hidden/masked or removed from logs.

Sensitive Code Example

With Signale log management framework the code is sensitive when an empty list of secrets is defined:

```
const { Signale } = require('signale');

const CREDIT_CARD_NUMBERS = fetchFromWebForm()
// here we suppose the credit card numbers are retrieved somewhere and CREDIT_CARD_NUMBERS looks like ["1234-5678-0000-9999", "1234-5678-0000-8888"]; fc

const options = {
  secrets: []         // empty list of secrets
};

const logger = new Signale(options); // Sensitive

CREDIT_CARD_NUMBERS.forEach(function(CREDIT_CARD_NUMBER) {
  logger.log('The customer ordered products with the credit card number = %s', CREDIT_CARD_NUMBER);
});
```

Compliant Solution

With Signale log management framework it is possible to define a list of secrets that will be hidden in logs:

```
const { Signale } = require('signale');

const CREDIT_CARD_NUMBERS = fetchFromWebForm()
// here we suppose the credit card numbers are retrieved somewhere and CREDIT_CARD_NUMBERS looks like ["1234-5678-0000-9999", "1234-5678-0000-8888"]; fc

const options = {
  secrets: ["([0-9]{4}-?)+"]
};

const logger = new Signale(options); // Compliant

CREDIT_CARD_NUMBERS.forEach(function(CREDIT_CARD_NUMBER) {
  logger.log('The customer ordered products with the credit card number = %s', CREDIT_CARD_NUMBER);
});
```

See

- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- CWE - CWE-532 - Insertion of Sensitive Information into Log File
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

assess_the_problem

Ask Yourself Whether

In a production environment:

- The web application uses confidential information and logs a significant amount of data.
- Logs are externalized to SIEM or Big Data repositories.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

With Signale log management framework the code is sensitive when an empty list of secrets is defined:

```
const { Signale } = require('signale');

const CREDIT_CARD_NUMBERS = fetchFromWebForm()
// here we suppose the credit card numbers are retrieved somewhere and CREDIT_CARD_NUMBERS looks like ["1234-5678-0000-9999", "1234-5678-0000-8888"]; fc

const options = {
  secrets: []         // empty list of secrets
};

const logger = new Signale(options); // Sensitive

CREDIT_CARD_NUMBERS.forEach(function(CREDIT_CARD_NUMBER) {
  logger.log('The customer ordered products with the credit card number = %s', CREDIT_CARD_NUMBER);
});
```

how_to_fix

Recommended Secure Coding Practices

Loggers should be configured with a list of confidential, personal information that will be hidden/masked or removed from logs.

Compliant Solution

With Signale log management framework it is possible to define a list of secrets that will be hidden in logs:

```
const { Signale } = require('signale');

const CREDIT_CARD_NUMBERS = fetchFromWebForm()
// here we suppose the credit card numbers are retrieved somewhere and CREDIT_CARD_NUMBERS looks like ["1234-5678-0000-9999", "1234-5678-0000-8888"]; fc

const options = {
  secrets: ["([0-9]{4}-?)+"]
};

const logger = new Signale(options); // Compliant

CREDIT_CARD_NUMBERS.forEach(function(CREDIT_CARD_NUMBER) {
  logger.log('The customer ordered products with the credit card number = %s', CREDIT_CARD_NUMBER);
});
```

See

- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- CWE - CWE-532 - Insertion of Sensitive Information into Log File
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

root_cause

Log management is an important topic, especially for the security of a web application, to ensure user activity, including potential attackers, is recorded and available for an analyst to understand what’s happened on the web application in case of malicious activities.

Retention of specific logs for a defined period of time is often necessary to comply with regulations such as GDPR, PCI DSS and others. However, to protect user’s privacy, certain informations are forbidden or strongly discouraged from being logged, such as user passwords or credit card numbers, which obviously should not be stored or at least not in clear text.

Repeated patterns in regular expressions should not match the empty string

Clave: javascript:S5842

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A regex should never include a repetitive pattern whose body would match the empty string. This is usually a sign that a part of the regex is redundant or does not do what the author intended.

Noncompliant code example

```
/(?:)*/      // same as the empty regex, the '*' accomplishes nothing
/(?:|x)*/    // same as the empty regex, the alternative has no effect
/(?:x|)*/    // same as 'x*', the empty alternative has no effect
/(?:x*|y*)*/ // same as 'x*', the first alternative would always match, y* is never tried
/(?:x?)*/    // same as 'x*'
/(?:x?)+/    // same as 'x*'
```

Compliant solution

```
/x*/
```

Regular expressions should not be too complicated

Clave: javascript:S5843

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Regular expressions
- MDN web docs - `RegExp`
- MDN web docs - Disjunction: `|`
- MDN web docs - Quantifiers
- MDN web docs - Lookahead assertion: `(?=...)`, `(?!...)`
- MDN web docs - Lookbehind assertion: `(?<=...)`, `(?<!...)`
- MDN web docs - Character classes
- MDN web docs - Groups and backreferences

root_cause

A complex regular expression is one that exhibits several or all of the following characteristics. It can be quite lengthy, containing multiple nested or repeated groups, numerous alternations, extensive use of backreferences and escape characters, lookaheads, lookbehinds, and other advanced features. Additionally, complex regular expressions may lack proper comments and documentation, making them challenging to comprehend and maintain. Overly complicated regular expressions are hard to read and maintain and can easily cause hard-to-find bugs.

To determine the complexity of a regular expression, each of the following operators increases the complexity by an amount equal to the current nesting level and also increases the current nesting level by one for its arguments:

- Disjunctions (`|`): when multiple `|` operators are used together, the subsequent ones only increase the complexity by 1
- Quantifiers (`*`, `+`, `?`, `{n,m}`, `{n,}` or `{n}`)
- Lookahead and lookbehind assertions

Additionally, each use of a character class and backreferences increase the complexity by 1 regardless of nesting.

This rule will raise an issue when total complexity is above the threshold `maxComplexity` (20 by default).

```
const datePattern = /^(?:(?:31(\/|-|\.)(?:0?[13578]|1[02]))\1|(?:(?:29|30)(\/|-|\.)(?:0?[13-9]|1[0-2])\2))(?:(?:1[6-9]|[2-9]\d)?\d{2})$|^(?:29(\/|-|\.)0
if (dateString.match(datePattern)) {
    handleDate(dateString);
}
```

If a regex is too complicated, you should consider replacing (partially or completely) it with regular code. Alternatively, split it apart into multiple patterns. If a regular expression is split among multiple variables, the complexity is calculated for each variable individually, not for the whole regular expression.

```
const datePattern = /^\d{1,2}([-/.])\d{1,2}\1\d{1,4}$/;
if (dateString.match(datePattern)) {
    const dateParts = dateString.split(/[-/.]/);
    const day = parseInt(dateParts[0]);
    const month = parseInt(dateParts[1]);
    const year = parseInt(dateParts[2]);
    // Put logic to validate and process the date based on its integer parts here
}
```

Names of regular expressions named groups should be used

Clave: javascript:S5860

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When using regular expressions, a capturing group provides extra information about the matched pattern. Named groups will store the matched contents on the `groups` property of the returned matches.

```
const regex = /(?<month>[0-9]{2})\/(?<year>[0-9]{2})/;
const { groups: {month, year} } = regex.exec("01/02"); // month is '01', year is '02'
```

This rule raises issues in several cases:

- Named groups are defined but never called by their name. To make the code more readable, remove unused named capturing groups.

```
const score = "14:1";
const scorePattern = /(?<player1>[0-9]+):(?<player2>[0-9]+)/; // Noncompliant - named groups are never used

if (scorePattern.exec(score)) {
  checkScore(score);
}
```

- Named groups are defined but called by their index. To make the code more readable:
  - access the matched contents using the named group; or
  - remove it from the pattern.

```
const datePattern = /(?<month>[0-9]{2})/(?<year>[0-9]{2})/;
const dateMatcher = datePattern.exec("01/02");

if (dateMatcher) {
  checkValidity(dateMatcher[1], dateMatcher[2]);  // Noncompliant - group indexes are used instead of names
}
```

- Named groups are referenced while not defined. This can have undesired effects, as it will return `undefined` if there are other named groups. If there are none, `groups` will be `undefined`, and trying to access the named group will throw a `TypeError` if its existence has not been checked.

```
const datePattern = /(?<month>[0-9]{2})/(?<year>[0-9]{2})/;
const dateMatcher = datePattern.exec("01/02");

if (dateMatcher) {
  checkValidity(dateMatcher.groups.day); // Noncompliant - there is no group called "day", returns `undefined`
}
```

resources

Documentation

- MDN web docs - Regular expressions
- MDN web docs - Groups and backreferences

Assertions should not be given twice the same argument

Clave: javascript:S5863

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

Assertions are statements that check whether certain conditions are true. They are used to validate that the actual results of a code snippet match the expected outcomes. By using assertions, developers can ensure that their code behaves as intended and identify potential bugs or issues early in the development process.

In Chai.js, there is no inherent problem with giving the same argument twice in an assertion. It won’t cause any errors or issues in the test execution itself. The test will still run and pass as long as the assertion is correct.

However, having the same argument twice in an assertion might indicate a design issue or a potential mistake in your test. In most cases, you don’t need to compare a variable to itself in a test, as it doesn’t provide any meaningful validation and is likely to be a bug due to the developer’s carelessness.

This rule raises an issue when a Chai assertion is given twice the same argument.

```
const assert = require('chai').assert;

describe("test the same object", function() {
    it("uses chai 'assert'", function() {
        const expected = '1';
        const actual = (1).toString();
        assert.equal(actual, actual); // Noncompliant: Asserting the same argument
    });
});
```

Make sure that the arguments of your assertions are not the same.

```
const assert = require('chai').assert;

describe("test the same object", function() {
    it("uses chai 'assert'", function() {
        const expected = '1';
        const actual = (1).toString();
        assert.equal(actual, expected);
    });
});
```

resources

Documentation

- Chai.js Documentation - API Reference

Regular expressions using Unicode character classes or property escapes should enable the unicode flag

Clave: javascript:S5867

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

JavaScript regular expressions provide Unicode character classses and Unicode property escapes for matching characters based on their Unicode values and Unicode properties respectively. When using Unicode property escapes like `\p{Alpha}` without the `u` flag, the regular expression will not match alphabetic characters but rather the `'\p{Alpha}'` string literal, which is likely a mistake.

This rules raises an issue when Unicode character classses and Unicode property escapes are used without the `u` flag.

Noncompliant code example

```
/\u{1234}/
/\p{Alpha}/
```

Compliant solution

```
/\u{1234}/u
/\p{Alpha}/u
```

Unicode Grapheme Clusters should be avoided inside regex character classes

Clave: javascript:S5868

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When placing Unicode Grapheme Clusters (characters which require to be encoded in multiple Code Points) inside a character class of a regular expression, this will likely lead to unintended behavior.

For instance, the grapheme cluster c̈ requires two code points: one for `'c'`, followed by one for the *umlaut* modifier `'\u{0308}'`. If placed within a character class, such as `[c̈]`, the regex will consider the character class being the enumeration `[c\u{0308}]` instead. It will, therefore, match every `'c'` and every *umlaut* that isn't expressed as a single codepoint, which is extremely unlikely to be the intended behavior.

This rule raises an issue every time Unicode Grapheme Clusters are used within a character class of a regular expression.

Noncompliant code example

```
"cc̈d̈d".replace(/[c̈d̈]/g, "X"); // result is "XXXXXX" and not expected "cXXd"
```

Compliant solution

```
"cc̈d̈d".replace(/c̈|d̈/g, "X"); // result is "cXXd"
```

Character classes in regular expressions should not contain the same character twice

Clave: javascript:S5869

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Remove the extra character, character range, or character class escape.

Noncompliant code example

```
/[0-99]/ // Noncompliant, this won't actually match strings with two digits
/[0-9.-_]/ // Noncompliant, .-_ is a range that already contains 0-9 (as well as various other characters such as capital letters)
/[a-z0-9\d]/ // Noncompliant, \d matches a digit and is equivalent to [0-9]
```

Compliant solution

```
/[0-9]{1,2}/
/[0-9.\-_]/
/[a-z\d]/
```

resources

Documentation

- MDN web docs - Character classes
- MDN web docs - Character class escape

root_cause

Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If the same character is listed twice in the same character class or if the character class contains overlapping ranges, this has no effect.

Thus duplicate characters in a character class are either a simple oversight or a sign that a range in the character class matches more than is intended or that the author misunderstood how character classes work and wanted to match more than one character. A common example of the latter mistake is trying to use a range like `[0-99]` to match numbers of up to two digits, when in fact it is equivalent to `[0-9]`. Another common cause is forgetting to escape the `-` character, creating an unintended range that overlaps with other characters in the character class.

Character ranges can also create duplicates when used with character class escapes. These are a type of escape sequence used in regular expressions to represent a specific set of characters. They are denoted by a backslash followed by a specific letter, such as `\d` for digits, `\w` for word characters, or `\s` for whitespace characters. For example, the character class escape `\d` is equivalent to the character range `[0-9]`, and the escape `\w` is equivalent to `[a-zA-Z0-9_]`.

A new session should be created during user authentication

Clave: javascript:S5876

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

introduction

An attacker may trick a user into using a predetermined session identifier. Consequently, this attacker can gain unauthorized access and impersonate the user’s session. This kind of attack is called session fixation, and protections against it should not be disabled.

root_cause

Session fixation attacks take advantage of the way web applications manage session identifiers. Here’s how a session fixation attack typically works:

- When a user visits a website or logs in, a session is created for them.
- This session is assigned a unique session identifier, stored in a cookie, in local storage, or through URL parameters.
- In a session fixation attack, an attacker tricks a user into using a predetermined session identifier controlled by the attacker. For example, the attacker sends the victim an email containing a link with this predetermined session identifier.
- When the victim clicks on the link, the web application does not create a new session identifier but uses this identifier known to the attacker.
- At this point, the attacker can hijack and impersonate the victim’s session.

What is the potential impact?

Session fixation attacks pose a significant security risk to web applications and their users. By exploiting this vulnerability, attackers can gain unauthorized access to user sessions, potentially leading to various malicious activities. Some of the most relevant scenarios are the following:

Impersonation

Once an attacker successfully fixes a session identifier, they can impersonate the victim and gain access to their account without providing valid credentials. This can result in unauthorized actions, such as modifying personal information, making unauthorized transactions, or even performing malicious activities on behalf of the victim. An attacker can also manipulate the victim into performing actions they wouldn't normally do, such as revealing sensitive information or conducting financial transactions on the attacker's behalf.

Data Breach

If an attacker gains access to a user's session, they may also gain access to sensitive data associated with that session. This can include personal information, financial details, or any other confidential data that the user has access to within the application. The compromised data can be used for identity theft, financial fraud, or other malicious purposes.

Privilege Escalation

In some cases, session fixation attacks can be used to escalate privileges within a web application. By fixing a session identifier with higher privileges, an attacker can bypass access controls and gain administrative or privileged access to the application. This can lead to unauthorized modifications, data manipulation, or even complete compromise of the application and its underlying systems.

how_to_fix

Upon user authentication, it is crucial to regenerate the session identifier to prevent fixation attacks. Passport provides a mechanism to achieve this by using the `req.session.regenerate()` method. By calling this method after successful authentication, you can ensure that each user is assigned a new and unique session ID.

Noncompliant code example

```
app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login' }),
  function(req, res) {
    // Noncompliant - no session.regenerate after login
    res.redirect('/');
  });
```

Compliant solution

```
app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login' }),
  function(req, res) {
    let prevSession = req.session;
    req.session.regenerate((err) => {
      Object.assign(req.session, prevSession);
      res.redirect('/');
    });
  });
```

How does this work?

The protection works by ensuring that the session identifier, which is used to identify and track a user's session, is changed or regenerated during the authentication process.

Here's how session fixation protection typically works:

1. When a user visits a website or logs in, a session is created for them. This session is assigned a unique session identifier, which is stored in a cookie or passed through URL parameters.
2. In a session fixation attack, an attacker tricks a user into using a predetermined session identifier controlled by the attacker. This allows the attacker to potentially gain unauthorized access to the user's session.
3. To protect against session fixation attacks, session fixation protection mechanisms come into play during the authentication process. When a user successfully authenticates, this mechanism generates a new session identifier for the user's session.
4. The old session identifier, which may have been manipulated by the attacker, is invalidated and no longer associated with the user's session. This ensures that any attempts by the attacker to use the fixed session identifier are rendered ineffective.
5. The user is then assigned the new session identifier, which is used for subsequent requests and session tracking. This new session identifier is typically stored in a new session cookie or passed through URL parameters.

By regenerating the session identifier upon authentication, session fixation protection helps ensure that the user's session is tied to a new, secure identifier that the attacker cannot predict or control. This mitigates the risk of an attacker gaining unauthorized access to the user's session and helps maintain the integrity and security of the application's session management process.

resources

Documentation

- Express.js Documentation - express-session

Articles & blog posts

- Fixing Session Fixation

Standards

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- OWASP Sesssion Fixation
- CWE - CWE-384 - Session Fixation
- STIG Viewer - Application Security and Development: V-222579 - Applications must use system-generated session identifiers that protect against session fixation.
- STIG Viewer - Application Security and Development: V-222582 - The application must not re-use or recycle session IDs.

Tests should check which exception is thrown

Clave: javascript:S5958

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Node.js Documentation - Assert
- Chai.js Documentation - Assert
- Chai.js Documentation - `expect` and `should`

root_cause

Assertions are statements that check whether certain conditions are true. They are used to validate that the actual results of a code snippet match the expected outcomes. By using assertions, developers can ensure that their code behaves as intended and identify potential bugs or issues early in the development process.

When the unit test is executed, the assertions are evaluated. If all the assertions in the test pass, it means the unit is functioning correctly for that specific set of inputs. If any of the assertions fail, it indicates that there is a problem with the unit’s implementation, and the test case helps identify the issue.

It is not good enough to test if an exception is raised, without checking which exception it is. Such tests will not be able to differentiate the expected exception from an unexpected one.

This rule raises an issue in the following cases:

- When an asynchronous Mocha test calls the `done()` callback, without parameters, in a `catch` block, and there is no reference to the caught exception in this block. Either the error should be passed to `done()` or the exception should be checked further.
- When Chai assertions are used to test if a function throws any exception, or an exception of type `Error` without checking the message.
- When Chai assertions are used to test if a function does not throw an exception of type `Error` without checking the message.

This rule doesn’t raise an issue when an assertion is negated. In such a case, the exception doesn’t need to be specific.

```
const expect = require("chai").expect;
const fs = require("fs");

describe("exceptions are not tested properly", function() {
    const funcThrows = function () { throw new TypeError('What is this type?'); };
    const funcNoThrow = function () { /*noop*/ };

    it("forgot to pass the error to 'done()'", function(done) {
        fs.readFile("/etc/zshrc", 'utf8', function(err, data) {
            try {
                expect(data).to.match(/some expected string/);
            } catch (e) {
                done(); // Noncompliant: either the exception should be passed to done(e), or the exception should be tested further.
            }
        });
    });

    it("does not 'expect' a specific exception", function() {
        expect(funcThrows).to.throw(); // Noncompliant: the exception should be tested.
        expect(funcThrows).to.throw(Error); // Noncompliant: the exception should be tested further.
    });
});
```

```
Tests should instead validate the exception message and/or type. By checking for the specific exception that is expected to be thrown, the test case car
```

```
const expect = require("chai").expect;
const { AssertionError } = require('chai');
const fs = require("fs");

describe("exceptions are tested properly", function() {
    const funcThrows = function () { throw new TypeError('What is this type?'); };
    const funcNoThrow = function () { /*noop*/ };

    it("did not forget to pass the error to 'done()'", function(done) {
        fs.readFile("/etc/zshrc", 'utf8', function(err, data) {
            try {
                expect(data).to.match(/some expected string/);
            } catch (e) {
                expect(e).to.be.an.instanceof(AssertionError);
                done();
            }
        });
    });

    it("does 'expect' a specific exception", function() {
        expect(funcThrows).to.throw(TypeError);
        expect(funcNoThrow).to.not.throw(Error, /My error message/);
    });
});
```

Extra boolean casts should be removed

Clave: javascript:S6509

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Boolean coercion
- MDN web docs - Type coercion
- MDN web docs - Truthy
- MDN web docs - Falsy

Articles & blog posts

- Alex Devero, How Truthy and Falsy Values in JavaScript Work

root_cause

In JavaScript, every value can be coerced into a boolean value: either `true` or `false`. Values that are coerced into `true` are said to be *truthy*, and those coerced into `false` are said to be *falsy*.

Explicit conversion to a boolean can be done with double negation (`!!`) or a `Boolean` call. Depending on the context, this may be redundant as JavaScript uses implicit type coercion and automatically converts values to booleans when used with logical operators, conditional statements, or any boolean context.

```
if (!!foo) { // Noncompliant: redundant '!!'
    // ...
}

if (Boolean(foo)) {  // Noncompliant: redundant 'Boolean' call
    // ...
}
```

A redundant boolean cast affects code readability. Not only the condition becomes more verbose but it also misleads the reader who might question the intent behind the extra cast. The condition can be written without the Boolean cast.

```
if (foo) {
    // ...
}
```

Users should not use internal APIs

Clave: javascript:S6627

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The public API of a framework, plugin, or library is the way its provider intended it to be used. API stability and compatibility (within the same major version number) of a library are guaranteed only for its public API.

Internal APIs are mere implementation details and are prone to breaking changes as the implementation of the library changes. No guarantees are being made about them. Therefore, users should not use internal APIs, even when visible.

What is the potential impact?

Code Stability

If not fixed, your code might break when the library is upgraded to a new version, even if only the minor version number or the patch number changes.

how_to_fix

Replace internal API usage with the public API designed for your use case. This may imply a refactoring of the affected code if no one-to-one replacement is available in the public API. If a specific functionality is required, copying the required parts of the implementation into your code may even be better than using the internal API.

Noncompliant code example

```
import { _parseWith } from './node_modules/foo/helpers'
```

Compliant solution

```
import { parse } from 'foo'
```

Constructors should not return values

Clave: javascript:S6635

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

introduction

Constructors should not return values.

resources

Documentation

- MDN web docs - constructor

how_to_fix

Do not return anything from the constructor.

Noncompliant code example

```
class TextMessage {
    constructor(msg) {
        this.text = msg;
        return msg; // Noncompliant
    }
}
```

Compliant solution

```
class TextMsg1 {
    constructor(msg) {
        this.text = msg;
    }
}


class TextMsg2 {
    constructor(msg) {
        if (!msg) {
            return; // ok to return nothing for flow control
        }
        this.text = msg;
    }
}
```

root_cause

JavaScript allows returning a value from a class constructor. This obscure feature is rarely used and is more likely a bug than the developer’s intention.

Unnecessary calls to ".bind()" should not be used

Clave: javascript:S6637

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `Function.prototype.bind()`

introduction

Unnecessary calls to `.bind()` should not be used.

how_to_fix

Remove calls to `.bind()` method.

Noncompliant code example

```
let x = function fn() {
    return 123;
}.bind({value: 456}); // Noncompliant


let y = (() => this.body).bind(document); // Noncompliant
```

Compliant solution

```
let x = (function callback() {
    return this.body;
}).bind(document); // ok, not an arrow function


let y = (function print(x) {
    console.log(x);
}).bind(this, foo); // ok, binds argument
```

root_cause

The `.bind()` method allows specifying the value of `this` and, optionally, the values of some function arguments. However, if `this` is not used in the function body, calls to `.bind()` do nothing and should be removed.

Calling `.bind()` on arrow functions is a bug because the value of `this` does not change when `.bind()` is applied to arrow functions.

Binary expressions should not always return the same value

Clave: javascript:S6638

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Review the code around the issue to find out why the expression always produces the same result. Pay attention to the operator precedence, comparing objects of different types, and comparing objects by reference (not by value!).

Noncompliant code example

```
!foo == null;
a + b ?? c;
x === [];
(foo=0) && bar;
```

Compliant solution

```
foo != null;
a + (b ?? c);
x.length === 0;
```

introduction

Comparisons that always evaluate to true or to false, logical expressions that either always or never short-circuit and comparisons to a newly constructed object should not be used.

root_cause

An expression that always produces the same result, regardless of the inputs, is unnecessary and likely indicates a programmer’s error. This can come from

- confusing operator precedence
- expecting strict equality between different types
- expecting objects to be compared by value
- expecting empty objects to be `false` or `null`
- mistyping `>=` for ⇒

This can also happen when you put an assignment in a logical sub-expression. While not strictly a bug, this practice is confusing and should be avoided.

In React "this.state" should not be mutated directly

Clave: javascript:S6746

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

React does not always propagate component state changes to the underlying DOM elements immediately. If you modify the state during an already ongoing update cycle, the change may be delayed until the next update. React tries to keep `this.state` in sync with what is currently in the DOM, so you should never directly modify `this.state` yourself. Instead, use the asynchronous `setState` method instead, which allows React to properly manage the current state, trigger the new update cycle, or batch the updates together if necessary.

Note that `setState()` is a *request* to change the state, not an immediate update. For example, if multiple components are changing the state in response to a user event, React will wait until the event handler has finished executing and then render all the changes in a single update. In other cases, the updates may be executed one at a time, so you should never make assumptions about how the component state will change in response to your request.

If your next state is a function of the current state, you should pass an updater function to the `setState()` that will give you access to the correct component state at the time of the execution.

The only place where you should directly modify the state is during the component initialization in a constructor function.

```
class MyComponent extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0
    };
  }

  incrementCount() {
    this.state.count++; // Noncompliant: direct mutation of state object
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

To fix this code use `setState()` method instead of directly mutating the state.

```
class MyComponent extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0
    };
  }

  incrementCount() {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

resources

Documentation

- React Documentation - `setState()` method
- React Documentation - Managing state

JSX elements should not use unknown properties and attributes

Clave: javascript:S6747

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Replace any unknown property or attribute with a known one, or add it to the list of exceptions.

Noncompliant code example

```
class Welcome extends React.Component {
  render() {
    return <div class="hello">Hello, World!</div>; // Noncompliant: 'class' is a reserved keyword in JavaScript
  }
}
```

Compliant solution

```
class Welcome extends React.Component {
  render() {
    return <div className="hello">Hello, World!</div>;
  }
}
```

Noncompliant code example

```
const Image = <img source={myImage} />; // Noncompliant: The 'img' tag does not recognize any 'source' attribute
```

Compliant solution

```
const Image = <img src={myImage} />;
```

resources

Documentation

- MDN web docs - HTML attribute reference
- MDN web docs - HTML global attributes
- MDN web docs - SVG attribute reference
- MDN web docs - ARIA states and properties
- MDN web docs - data-*
- React Documentation - Standard DOM props

root_cause

React components often render HTML elements, and developers can pass various props (properties) to these elements. However, React has its own set of supported properties and attributes, and it’s essential to avoid using unknown or invalid properties when working with such elements to prevent unexpected behavior at runtime.

The rule reports any instances where you are using a property or attribute that is not recognized by React or the HTML element you are rendering.

React "children" should not be passed as prop

Clave: javascript:S6748

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Passing Props to a Component

root_cause

When using JSX the component children should be passed between opening and closing tags. Passing children in a `children` prop may work sometimes, but will lead to errors if children are passed both as nested components and `children` prop at the same time.

When not using JSX, the children should be passed to `createElement()` method as extra arguments after the `props` object.

```
<div children='Children' />
<Foo children={<Bar />} />

React.createElement("div", { children: 'Children' })
```

To fix the code, remove the `children` prop and pass the children between opening and closing JSX tags or as extra arguments to `createElement()` function.

```
<div>Children</div>
<Foo><Bar /></Foo>

React.createElement("div", {}, 'Children');
```

Redundant React fragments should be removed

Clave: javascript:S6749

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Fragments

root_cause

React fragments are a feature in React that allows you to group multiple elements together without adding an extra DOM element. They are a way to return multiple elements from a component’s render method without requiring a wrapping parent element.

However, a fragment is redundant if it contains only one child, or if it is the child of an HTML element.

```
<><Foo /></>;    // Noncompliant: The fragment has only one child
<p><>foo</></p>; // Noncompliant: The fragment is the child of the HTML element 'p'
```

You can safely remove the redundant fragment while preserving the original behaviour.

```
<Foo />;
<p>foo</p>;
```

The return value of "ReactDOM.render" should not be used

Clave: javascript:S6750

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In React, the `ReactDOM.render()` method is used to render a React component into a DOM element. It has a return value, but it’s generally recommended not to use it. The method might return a reference to the root `ReactComponent` instance, but it can be unpredictable and may not always be useful. Indeed, the return value can vary depending on the version of React you’re using and the specific circumstances in which it’s called.

```
const instance = ReactDOM.render(<App />, document.body); // Noncompliant: using the return value of 'ReactDOM.render'
doSomething(instance);
```

```
ReactDOM.render(<App />, document.body);
```

Alternatively, if you really need a reference to the root `ReactComponent` instance, the preferred solution is to attach a "callback ref" to the root element.

```
ReactDOM.render(<App />, document.body, callbackRef);
```

resources

Documentation

- React Documentation - ReactDom#render
- React Documentation - Adding a ref to your component

The return value of "useState" should be destructured and named symmetrically

Clave: javascript:S6754

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - useState

root_cause

In React, `useState` is a hook that allows functional components to manage and update state in a manner similar to class components. When you use the `useState` hook, it returns an array with two values: the current state value and a function to update that state value.

Destructuring these values and naming them symmetrically (i.e., using consistent variable names for both the current state and the update function) is a recommended best practice:

- When you destructure and name the values symmetrically, it makes your code more readable and self-explanatory. Other developers can quickly understand the purpose of each variable without needing to refer back to the `useState` function call.
- Following a naming convention where the state variable and its corresponding update function have similar names is a common practice in the React community. It helps maintain consistency and makes it easier for others to understand your code.
- If you don’t name the variables symmetrically, it can lead to confusion, especially in larger components or when multiple state variables are involved. You might accidentally use the wrong variable when updating the state, which can result in bugs that are hard to track down.

```
import { useState } from 'react';
function MyComponent() {
  const [count, update] = useState(0); // Noncompliant
  return <div onClick={() => update(count + 1)}>{count}</div>
}
```

You should destructure the return value of `useState` calls in terms of the current state and a function to update that state and name them symmetrically.

```
import { useState } from 'react';
function MyComponent() {
  const [count, setCount] = useState(0);
  return <div onClick={() => setCount(count + 1)}>{count}</div>
}
```

"setState" should use a callback when referencing the previous state

Clave: javascript:S6756

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In React, calling `setState` is the primary way to update a component’s state. However, calling `setState` is often asynchronous. React batches state updates for performance reasons, which means that when you call `setState`, React doesn’t immediately update the state and trigger a re-render. Instead, it schedules the update for later, and multiple `setState` calls within the same event handler or function may be batched together. This can lead to unexpected behavior if you assume that state updates are immediate. Therefore, you should not rely on their values for calculating the next state.

```
function increment() {
  this.setState({count: this.state.count + 1}); // Noncompliant
}
```

To mitigate this, you should use functional updates when the new state depends on the previous state, ensuring that you’re always working with the latest state. This can be done with a second form of `setState` that accepts a function rather than an object.

```
function increment() {
  this.setState(prevState => ({count: prevState.count + 1}));
}
```

resources

Documentation

- React Documentation - setState
- React Documentation - setState caveats

"this" should not be used in functional components

Clave: javascript:S6757

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Defining a Component
- React Documentation - Passing Props to a Component
- React Documentation - Legacy class components

root_cause

Referring to `this` in React functional components would be an error because the components are just regular JavaScript functions and do not have an object associated with them. Functional components receive their props as a first argument to the component function, so you can access them directly, and it is a common practice to destructure them right away.

```
function UserProfile({firstName, lastName}){
    return (
        <div className="user">{firstName} {lastName}</div>
    );
}
```

React also supports legacy class-based components, where `this` keyword refers to the component instance object, but this style of writing components is no longer recommended, and mixing it with functional components will lead to errors.

```
function MyComponent(props){
    const foo = this.props.bar; // Noncompliant: remove 'this'
    return (
        <div>{foo}</div>
    );
}
```

To fix the issue, remove references to `this` from your functional component code. Make also sure you are not mixing functional and class-based component styles.

```
function MyComponent({bar}){
    const foo = bar;
    return (
        <div>{foo}</div>
    );
}
```

DOM elements with ARIA roles should have the required properties

Clave: javascript:S6807

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

Each role in ARIA has a set of required attributes that must be included for the role to be properly understood by assistive technologies. These attributes are known as "required aria-* properties".

For example, if an element has a role of "checkbox", it must also include the aria-checked property. This property indicates whether the checkbox is checked (true), unchecked (false), or in a mixed state (mixed).

This rule checks that each element with a defined ARIA role also has all required attributes.

how_to_fix

Check that each element with a defined ARIA role also has all required attributes.

```
<div role="checkbox">Unchecked</div> {/* Noncompliant: aria-checked is missing */}
```

To fix the code add missing aria-* attributes.

```
<div role="checkbox" aria-checked={isChecked}>Unchecked</div>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)
- MDN web docs - ARIA states and properties (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

DOM elements with ARIA role should only have supported properties

Clave: javascript:S6811

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA properties, also known as "aria-* properties", are special attributes used in HTML to enhance the accessibility of web elements. They provide additional semantics to help assistive technologies, like screen readers, interpret the element.

Roles, on the other hand, define what an element is or does in the context of a web page. Some elements have explicit roles, which are directly defined by the developer. For example, a div element might be given a role of "button". Other elements have implicit roles, which are inferred based on the type of the element. For example, an anchor tag <a href="#" /> has an implicit role of "link".

This rule ensures that the ARIA properties used on an element are ones that are supported by the role of that element. For instance, the ARIA property `aria-required` is not supported by the role `link`. Therefore, using `aria-required` on an anchor tag would violate this rule.

```
<div role="checkbox" aria-chekd={isChecked}>Unchecked</div> {/* Noncompliant: aria-chekd is not supported */}
```

how_to_fix

Check the spelling of the aria-* attributes and verify that they are actually supported by the element role. Remove non-compatible attributes or replace them with the correct ones.

```
<div role="checkbox" aria-checked={isChecked}>Unchecked</div>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA states and properties (Reference)
- MDN web docs - ARIA roles (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

Prefer tag over ARIA role

Clave: javascript:S6819

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) roles are used to make web content and web applications more accessible to people with disabilities. However, you should not use an ARIA role on a generic element (like `span` or `div`) if there is a semantic HTML tag with similar functionality, just use that tag instead.

For example, instead of using a div element with a button role (`<div role="button">Click me</div>`), you should just use a button element (`<button>Click me</button>`).

Semantic HTML tags are generally preferred over ARIA roles for accessibility due to their built-in functionality, universal support by browsers and assistive technologies, simplicity, and maintainability. They come with inherent behaviors and keyboard interactions, reducing the need for additional JavaScript. Semantic HTML also enhances SEO by helping search engines better understand the content and structure of web pages. While ARIA roles are useful, they should be considered a last resort when no suitable HTML element can provide the required behavior or semantics.

```
<div role="button" onClick={handleClick} /* Noncompliant */>Click me</div>
```

Replace the ARIA role with an appropriate HTML tag.

```
<button onClick={handleClick}>Click me</button>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA roles (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

"case" and "default" clauses should not contain lexical declarations

Clave: javascript:S6836

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - switch

root_cause

The ECMAScript specification allows for creating block-level lexical declarations (`let`, `const`, `function`, and `class`) in any block statement or expression. However, when these declarations are made inside the `case` or `default` clause of a `switch` statement, they are not confined to the block of that `case` or `default` clause. Instead, they apply to the whole `switch` block but only get initialized when the cases are reached, which can lead to unexpected behavior.

```
switch (foo) {
    case 1:
        let x = 1; // Noncompliant
        break;
    case 2:
        const y = 2; // Noncompliant
        break;
    case 3:
        function f() {} // Noncompliant
        break;
    case 4:
        class C {} // Noncompliant
        break;
}
```

To fix this, you can create a nested block within each `case` or `default` clause, ensuring each declaration is properly scoped to its respective block.

```
switch (foo) {
    case 1: {
        let x = 1;
        break;
    }
    case 2: {
        const y = 2;
        break;
    }
    case 3: {
        function f() {}
        break;
    }
    case 4: {
        class C {}
        break;
    }
}
```

DOM elements should use the "autocomplete" attribute correctly

Clave: javascript:S6840

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Not providing autocomplete values in form fields can lead to content inaccessibility. The function of each standard input field, which gathers a person’s personal data, is systematically determined according to the list of 53 Input Purposes for User Interface Components. If the necessary autocomplete attribute values are absent, screen readers will not be able to identify and read these fields. This lack of information can hinder users, particularly those using screen readers, from properly navigating and interacting with forms.

For screen readers to operate effectively, it is imperative that the autocomplete attribute values are not only valid but also correctly applied.

how_to_fix

Ensure the autocomplete attribute is correct and suitable for the form field it is used with:

- Identify the input type: The autocomplete attribute should be used with form elements like `<input>`, `<select>`, and `<textarea>`. The type of input field should be clearly identified using the `type` attribute, such as `type="text"`, `type="email"`, or `type="tel"`.
- Specify the autocomplete value: The value of the autocomplete attribute should be a string that specifies what kind of input the browser should autofill. For example, `autocomplete="name"` would suggest that the browser autofill the user’s full name.
- Use appropriate autocomplete values: The value you use should be appropriate for the type of input. For example, for a credit card field, you might use `autocomplete="cc-number"`. For a country field in an address form, you might use `autocomplete="country"`.

For additional details, please refer to the guidelines provided in the HTML standard.

Noncompliant code example

```
function MyInput() {
    return <input type="text" autocomplete="foo" />; // Noncompliant
}
```

Compliant solution

```
function MyInput() {
    return <input type="text" autocomplete="name" />;
}
```

resources

Documentation

- WCAG - Identify Input Purpose
- WCAG - Input Purposes for User Interface Components
- HTML Standard - Enabling client-side automatic filling of form controls
- HTML Standard - Autofilling form controls: the autocomplete attribute

---

"tabIndex" values should be 0 or -1

Clave: javascript:S6841

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Positive `tabIndex` values can disrupt the natural tab order of the webpage. This can be confusing for screen reader users who rely on a logical tab order to navigate through the content. If the tab order doesn’t match the visual or logical order of elements, users may struggle to understand the page structure.

Therefore, it’s recommended to avoid using positive `tabIndex` values.

how_to_fix

If you need to make an element focusable that isn’t by default (like a `<div>` or `<span>`), you can use `tabIndex="0"`. This will add the element to the natural tab order based on its position in the HTML. Otherwise, either remove the `tabIndex` value or use `tabIndex="-1"` to remove the element from the tab order.

Noncompliant code example

```
function MyDiv() {
    return (
        <div>
            <span tabIndex="5">foo</span> // Noncompliant
            <span tabIndex="3">bar</span> // Noncompliant
            <span tabIndex="1">baz</span> // Noncompliant
            <span tabIndex="2">qux</span> // Noncompliant
        </div>
    );
}
```

Compliant solution

```
function MyDiv() {
    return (
        <div>
            <span tabIndex="0">foo</span>
            <span tabIndex="-1">bar</span>
            <span tabIndex={0}>baz</span>
            <span>qux</span>
        </div>
    );
}
```

resources

Documentation

- MDN web docs - tabindex
- WCAG - Focus Order

Non-interactive DOM elements should not have interactive ARIA roles

Clave: javascript:S6842

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Ensure that non-interactive DOM elements do not use interactive ARIA roles.

Noncompliant code example

```
function myListElement() {
    return <li role="button">Foo</li>; // Noncompliant; "li" isn't interactive, but "button" is
}
```

Compliant solution

```
function myListElement() {
    return <li role="listitem">Foo</li>;
}
```

resources

Documentation

- WCAG - Name, Role, Value
- WCAG - WAI-ARIA Roles
- MDN web docs - WAI-ARIA Roles

root_cause

Non-interactive DOM elements are those that do not have built-in interactivity or do not require user interaction. Examples include `<div>`, `<p>`, `<img>`, `<h1>` to `<h6>`, and `<ul>`, among others. These elements are typically used to structure content and layout but do not have any inherent interactive behavior.

Interactive ARIA roles, on the other hand, are used to make elements interactive and accessible. These roles include `button`, `link`, `checkbox`, `menuitem`, `tab`, and others. They are used to communicate the type of interaction that users can expect from an element.

Non-interactive DOM elements should not use interactive ARIA roles because it can confuse assistive technologies and their users. For example, if a `<div>` (a non-interactive element) is given an interactive role like "button", assistive technologies like screen readers will announce it as a button. However, since `<div>` doesn’t have the inherent behavior of a button, it can confuse users who expect it to behave like a button when interacted with.

This can lead to a poor user experience and can make the website less accessible for users relying on assistive technologies. Therefore, it’s important to ensure that non-interactive DOM elements are not given interactive ARIA roles.

Interactive DOM elements should not have non-interactive ARIA roles

Clave: javascript:S6843

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Ensure that interactive DOM elements are not given non-interactive ARIA roles.

Noncompliant code example

```
function myButton() {
    return <button role="article">Click me!</button>; // Noncompliant; "button" is interactive, but "article" isn't
}
```

Compliant solution

```
function myButton() {
    return <button role="button">Click me!</button>;
}
```

root_cause

Interactive DOM elements are elements that users can interact with. These include buttons, links, form inputs, and other elements that can be clicked, focused, or otherwise manipulated by the user. ARIA roles, on the other hand, are used to improve accessibility by providing additional semantic information about an element’s purpose and behavior. ARIA roles can be divided into two categories: interactive roles and non-interactive roles.

Interactive ARIA roles are used for elements that a user can interact with, such as buttons or sliders. Non-interactive ARIA roles are used for elements that are not meant to be interacted with, such as content containers or landmarks. Examples of non-interactive ARIA roles include `article`, `banner`, `complementary`, `contentinfo`, `definition`, `directory`, `document`, `feed`, `figure`, `group`, `heading`, `img`, `list`, `listitem`, `math`, `none`, `note`, `presentation`, `region`, `separator`, `status`, `term`, and `tooltip`.

Interactive DOM elements should not have non-interactive ARIA roles because it can confuse assistive technologies and their users. For example, if a button (an interactive element) is given a non-interactive ARIA role like `article`, it can mislead users into thinking that the button is just a piece of content, not something they can interact with. This can lead to a poor user experience, especially for users who rely on assistive technologies to navigate the web.

Therefore, it’s important to ensure that interactive DOM elements are not given non-interactive ARIA roles.

resources

Documentation

- WCAG - Name, Role, Value
- WCAG - WAI-ARIA Roles
- MDN web docs - WAI-ARIA Roles

Anchor tags should not be used as buttons

Clave: javascript:S6844

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- WebAIM - Introduction to Links and Hypertext
- Links vs. Buttons in Modern Web Applications
- Using ARIA - Notes on ARIA use in HTML

how_to_fix

To fix this issue, you should use the appropriate HTML elements for their intended purposes. If you need to create a hyperlink, use the `<a>` tag with a `href` attribute. If you need to create a button, use the `<button>` tag.

Noncompliant code example

```
const MyComponent = () => {
    return <>
        <a href="javascript:void(0)" onClick={foo}>Perform action</a>
        <a href="#" onClick={foo}>Perform action</a>
        <a onClick={foo}>Perform action</a>
    </>;
};
```

Compliant solution

```
const MyComponent = () => {
    return <>
      <button onClick={foo}>Perform action</button>
      <a href="#section" onClick={foo}>Perform action</a>
    </>;
};
```

introduction

The `<a>` tag in HTML is designed to create hyperlinks, which can link to different sections of the same page, different pages, or even different websites. However, sometimes developers misuse `<a>` tags as buttons, which can lead to accessibility issues and unexpected behavior.

This rule checks that `<a>` tags are used correctly as hyperlinks and not misused as buttons. It verifies that each `<a>` tag has a `href` attribute, which is necessary for it to function as a hyperlink. If an `<a>` tag is used without a `href` attribute, it behaves like a button, which is not its intended use.

Using the correct HTML elements for their intended purpose is crucial for accessibility and usability. It ensures that the website behaves as expected and can be used by all users, including those using assistive technologies. Misusing HTML elements can lead to a poor user experience and potential accessibility violations.

Compliance with this rule will ensure that your HTML code is semantically correct, accessible, and behaves as expected.

root_cause

Misusing `<a>` tags as buttons can lead to several issues:

- Accessibility: Screen readers and other assistive technologies rely on the correct use of HTML elements to interpret and interact with the content. When `<a>` tags are used as buttons, it can confuse these technologies and make the website less accessible to users with disabilities.
- Usability: The behavior of `<a>` tags and buttons is different. For example, buttons can be triggered using the space bar, while `<a>` tags cannot. Misusing these elements can lead to unexpected behavior and a poor user experience.
- Semantic correctness: Each HTML element has a specific purpose and meaning. Using elements for purposes other than their intended use can make the code harder to understand and maintain.

- SEO implications: Search engines use the structure and semantics of HTML to understand and rank web pages. Misusing HTML elements can negatively impact a website's SEO.

Non-interactive DOM elements should not have the `tabIndex` property

Clave: javascript:S6845

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - tabindex
- The a11y project - Use the tabindex attribute

how_to_fix

Simply remove the `tabIndex` attribute or set it to `"-1"` to fix the issue.

Noncompliant code example

```
<div tabIndex="0" />
```

Compliant solution

```
<div />
```

introduction

Navigation using the Tab key should be restricted to elements on the page that users can interact with.

root_cause

The misuse of the `tabIndex` attribute can lead to several issues:

- Navigation Confusion: It can confuse users who rely on keyboard navigation, as they might expect to tab through interactive elements like links and buttons, not static content.
- Accessibility Issues: It can create accessibility problems, as assistive technologies provide their own page navigation mechanisms based on the HTML of the page. Adding unnecessary tabindexes can disrupt this.
- Increased Tab Ring Size: It unnecessarily increases the size of the page's tab ring, making navigation more cumbersome.

DOM elements should not use the "accesskey" property

Clave: javascript:S6846

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - accesskey
- MDN web docs - Accessibility concerns

root_cause

The `accesskey` attribute, despite its potential utility, is fraught with numerous issues that limit its effectiveness and usability:

- Screen readers' implementation of `accesskey` largely depends on the browser used, as they rely on browsers for much of their functionality. Some screen readers may repeatedly indicate the `accesskey` value each time the element is encountered, potentially causing unnecessary repetition and noise for the user.
- Conflicts between `accesskey` shortcuts and other keyboard shortcuts, such as those of browsers, operating systems, assistive technologies, or browser extensions, are frequent. This overlap can lead to uncertainty and potentially trigger unintended actions, causing user confusion.

- While keyboard shortcuts are vital for screen reader functionality, conflicts can disable either the screen reader or `accesskey` shortcuts. Typically, screen reader shortcuts take precedence, disabling the `accesskey` but preserving screen reader functionality. However, this can cause confusion for users attempting to activate an `accesskey`.
- No keystroke combinations can guarantee zero conflicts with all browsers, assistive technologies, or operating systems, particularly considering foreign languages. For instance, an `accesskey` shortcut that works in an English browser may conflict in the same browser set in another language due to different menu naming conventions.
- While using numerals instead of letters for keyboard shortcuts could reduce conflicts, it’s not a foolproof solution. There’s no standard correlation between numbers and web functions, which could lead to user confusion.
- Unlike the Windows environment that highlights keyboard shortcuts in menus, web pages or applications lack a standardized method to notify users about available `accesskey` shortcuts.

Given these concerns, it is generally recommended to avoid using `accesskeys`.

```
function div() {
    return <div accessKey="h" />;
}
```

Do not use `accesskeys` at all.

```
function div() {
    return <div />;
}
```

Mutable variables should not be exported

Clave: javascript:S6861

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, a mutable variable is one whose value can be changed after it has been initially set. This is in contrast to immutable variables, whose values cannot be changed once they are set.

Exporting mutable variables can lead to unpredictable behavior and bugs in your code. This is because any module that imports the variable can change its value. If multiple modules import and change the value of the same variable, it can become difficult to track what the current value of the variable is and which module changed it last.

how_to_fix

If the value of the variable does not need to change, you can declare it as a constant using the `const` keyword. Alternatively, if you have a group of related variables that need to be mutable, consider using a class to encapsulate them. You can then export an instance of the class, or a factory function that creates instances of the class.

Noncompliant code example

```
let mutableVar = "initial value";

export { mutableVar }; // Noncompliant
```

Compliant solution

```
const immutableVar = "constant value";
export { immutableVar };
```

or

```
class MyClass {
  constructor() {
    this.mutableVar = "initial value";
  }
}

export function createMyClass() {
  return new MyClass();
}
```

resources

Documentation

- MDN web docs - let
- MDN web docs - const
- MDN web docs - Mutable
- MDN web docs - Immutable

Deprecated React APIs should not be used

Clave: javascript:S6957

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - React Reference Overview

root_cause

Deprecated methods are functions or properties that are no longer recommended and are likely to be removed in future updates of the library. They are often replaced with newer methods that offer better performance, security, or usability.

Using deprecated methods in React can lead to the following issues:

- **Code Maintainability**: As deprecated methods are removed in future versions, the code will break if not updated. This can lead to increased time and effort in code maintenance.
- **Performance**: Newer methods often come with performance improvements. Using deprecated methods can lead to slower app performance.
- **Security**: Deprecated methods may have known security issues that are fixed in newer methods.

```
import React, { useState, useEffect } from 'react';

function MyComponent(props) {
  const [state, setState] = useState(initialState);

  componentWillReceiveProps(nextProps) { // Noncompliant: deprecated lifecycle method
    // Some code here...
  }

  componentWillUpdate(nextProps, nextState) { // Noncompliant: deprecated lifecycle method
    // Some code here...
  }

  render() {
    return <div>Hello World</div>;
  }
}
```

To fix this issue, check React’s upgrading guide, replace the deprecated methods with their newer counterparts, and adapt your component’s implementation accordingly.

```
import React, { useState, useEffect } from 'react';

function MyComponent(props) {
  const [state, setState] = useState(initialState);

  // Using useEffect to replace deprecated lifecycle methods
  useEffect(() => {
    // Code to run on component update
  }, [props]); // This will run when `props` changes

  return <div>Hello World</div>;
}
```

Literals should not be used as functions

Clave: javascript:S6958

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Functions
- MDN web docs - Template literals

root_cause

Calling a literal throws a TypeError, and is likely the result of an unintentional error in the code.

This rule raises an issue when an attempt is made to use a literal as a function.

```
true(); // Noncompliant, literal should not be used as function
```

This rule also detects when a literal is used as a *tag* function.

```
true``; // Noncompliant, literal should not be used as tag function
```

"Array.reduce()" calls should include an initial value

Clave: javascript:S6959

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `Array.prototype.reduce()` method in JavaScript is used to apply a function against an accumulator and each element in the array (from left to right) to reduce it to a single output value. It is a convenient method that can simplify logic in your code.

However, it’s important to always provide an initial value as the second argument to `reduce()`. The initial value is used as the first argument to the first call of the callback function. If no initial value is supplied, JavaScript will use the first element of the array as the initial accumulator value and start iterating at the second element.

This can lead to runtime errors if the array is empty, as `reduce()` will throw a TypeError.

```
function sum(xs) {
  return xs.reduce((acc, current) => acc + current); // Noncompliant
}
console.log(sum([1, 2, 3, 4, 5])); // Prints 15
console.log(sum([])); // TypeError: Reduce of empty array with no initial value
```

To fix this, always provide an initial value as the second argument to `reduce()`.

```
function sum(xs) {
  return xs.reduce((acc, current) => acc + current, 0);
}
console.log(sum([1, 2, 3, 4, 5])); // Prints 15
console.log(sum([])); // Prints 0
```

resources

Documentation

- MDN web docs - Array.prototype.reduce()
- MDN web docs - TypeError: Reduce of empty array with no initial value

---

Comma operator should not be used

Clave: javascript:S878

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Comma operator (,)

root_cause

The comma operator takes two expressions, executes them from left to right, and returns the result of the second one. The use of this operator is generally detrimental to the readability and reliability of code, and the same effect can be achieved by other means.

```
i = a += 2, a + b;  // Noncompliant: What's the value of i ?
```

Writing each expression on its own line will improve readability and might fix misunderstandings.

```
a += 2;
i = a + b; // We probably expected to assign the result of the addition to i, although the previous code wasn't doing it.
```

Exceptions

The comma operator is tolerated:

- In initializations and increment expressions of `for` loops.

```
for (i = 0, j = 5; i < 6; i++, j++) { ... }
```

- If the expression sequence is explicitly wrapped in parentheses.

```
i = (a += 2, a + b); // Compliant by exception
```

Increment (++) and decrement (--) operators should not be used in a method call or mixed with other operators in an expression

Clave: javascript:S881

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The use of increment and decrement operators in method calls or in combination with other arithmetic operators is not recommended, because:

- It can significantly impair the readability of the code.
- It introduces additional side effects into a statement, with the potential for undefined behavior.
- It is safer to use these operators in isolation from any other arithmetic operators.

Noncompliant code example

```
u8a = ++u8b + u8c--;
foo = bar++ / 4;
```

Compliant solution

The following sequence is clearer and therefore safer:

```
++u8b;
u8a = u8b + u8c;
u8c--;
foo = bar / 4;
bar++;
```

An open curly brace should be located at the end of a line

Clave: javascript:S1105

Severidad: MINOR

Estado: ⚠️ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when the brace-style is not respecting the convention setup in parameter:

- 1tbs (default)
- allman
- stroustrup

Noncompliant code example

Using the parameter default (1tbs):

```
if (condition)
{                                                      //Noncompliant
  doSomething();
}                                                      //Noncompliant
else {
  doSomethingElse();
}
```

Compliant solution

```
if (condition) {                                   //Compliant
  doSomething();
} else {                                           //Compliant
  doSomethingElse();
}
```

Exceptions

- Object literals appearing as arguments can start on their own line.

```
functionWithObject(
   {                                                 //Compliant
        g: "someValue"
   }
);
```

- When blocks are inlined (left and right curly braces on the same line), no issue is triggered.

```
if(condition) {doSomething();}                       //Compliant
```

"switch" statements should not contain non-case labels

Clave: javascript:S1219

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Even if it is legal, mixing case and non-case labels in the body of a switch statement is very confusing and can even be the result of a typing error.

Noncompliant code example

Case 1, the code is syntactically correct but the behavior is not the expected one

```
switch (day) {
  case MONDAY:
  case TUESDAY:
  WEDNESDAY:   // instead of "case WEDNESDAY"
    doSomething();
    break;
  ...
}
```

Case 2, the code is correct and behaves as expected but is hardly readable

```
switch (day) {
  case MONDAY:
    break;
  case TUESDAY:
    foo:for(i = 0 ; i < X ; i++) {
         /* ... */
        break foo;  // this break statement doesn't relate to the nesting case TUESDAY
         /* ... */
    }
    break;
    /* ... */
}
```

Compliant solution

Case 1

```
switch (day) {
  case MONDAY:
  case TUESDAY:
  case WEDNESDAY:
    doSomething();
    break;
  ...
}
```

Case 2

```
switch (day) {
  case MONDAY:
    break;
  case TUESDAY:
    compute(args); // put the content of the labelled "for" statement in a dedicated method
    break;

    /* ... */
}
```

Initial values of parameters, caught exceptions, and loop variables should not be ignored

Clave: javascript:S1226

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Functions
- MDN web docs - Catch binding
- MDN web docs - `for...of`
- MDN web docs - `for...in`
- Wikipedia - Call by sharing

root_cause

Ignoring function parameters or overwriting them with a new value without reading them can lead to confusion and errors in the code. Developers won’t be able to tell whether the original parameter or some temporary variable is being accessed without going through the whole function. It may indicate that the function is not properly designed or that there is a mistake in the code.

Moreover, some developers might also expect assignments of function parameters to be visible to callers, which is not the case. Arguments are always passed by value and never passed by reference. If a function reassigns a parameter, the value won’t change outside the function. It is not possible to simulate an assignment on that variable in the caller’s scope. However, objects are passed by value to their reference (passed by sharing), which means if the object’s properties are mutated, the change will impact the outside of the function.

The same logic applies to caught exceptions and variable declarations inside `for...in` and `for...of` statements: their initial values should not be ignored.

```
function myFunction(name, strings) {
  name = foo; // Noncompliant: initial value of 'name' is ignored

  for (let str of strings) {
    str = "";  // Noncompliant: initial value of 'str' is ignored
  }
}
```

Function parameters, caught exceptions, and variables initialized in `for...in` and `for...of`statements should be read at least once before reassigning them. If they do not need to be read, the code should be refactored to avoid confusion.

```
function myFunction(name, strings) {
  const nameCopy = name;
  name = foo;

  for (let str of strings) {
    const strCopy = str;
    str = "";
  }
}
```

Exceptions

There is a common pattern in JavaScript to overwrite certain parameters depending on other parameters that are optional. For example, a callback is, by convention, always passed in the last position. If a parameter in a previous position was not passed, the callback will be passed in its position instead.

Therefore, the rule ignores parameter reassignments that are inside an `if` statement block.

```
function myFunction(param, optionalParam, cb) {
  if (typeof optionalParam === 'function') {
    cb = optionalParam;
    optionalParam = {};
  }
}
```

"if" statements should be preferred over "switch" when simpler

Clave: javascript:S1301

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A `switch` statement is a control flow statement that allows you to execute different blocks of code based on the value of an expression. It provides a more concise way to handle multiple conditions compared to using multiple `if-else` statements.

If you only have a single condition to check, using an `if` statement is simpler and more concise. `switch` statements are designed for handling multiple cases, so using them for a single condition can be overkill and less readable.

This rule raises an issue when a `switch` statement has only one `case` clause and possibly a `default` one.

```
switch (condition) { // Noncompliant: The switch has only one case and a default
  case 0:
    doSomething();
    break;
  default:
    doSomethingElse();
    break;
}
```

Use a `switch` statement when you have multiple cases to handle and an `if` statement when you have only one condition to check.

```
if (condition === 0) {
  doSomething();
} else {
  doSomethingElse();
}
```

resources

Documentation

- MDN web docs - `switch`
- MDN web docs - `if...else`

Using hardcoded IP addresses is security-sensitive

Clave: javascript:S1313

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

The disclosed IP address is sensitive, e.g.:

- Can give information to an attacker about the network topology.
- It's a personal (assigned to an identifiable person) IP address.

There is a risk if you answered yes to any of these questions.

Sensitive Code Example

```
ip = "192.168.12.42"; // Sensitive

const net = require('net');
var client = new net.Socket();
client.connect(80, ip, function() {
  // ...
});
```

how_to_fix

Recommended Secure Coding Practices

Don’t hard-code the IP address in the source code, instead make it configurable with environment variables, configuration files, or a similar approach. Alternatively, if confidentially is not required a domain name can be used since it allows to change the destination quickly without having to rebuild the software.

Compliant Solution

```
ip = process.env.IP_ADDRESS; // Compliant

const net = require('net');
var client = new net.Socket();
client.connect(80, ip, function() {
  // ...
});
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

default

Hardcoding IP addresses is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2006-5901
- CVE-2005-3725

Today’s services have an ever-changing architecture due to their scaling and redundancy needs. It is a mistake to think that a service will always have the same IP address. When it does change, the hardcoded IP will have to be modified too. This will have an impact on the product development, delivery, and deployment:

- The developers will have to do a rapid fix every time this happens, instead of having an operation team change a configuration file.
- It misleads to use the same address in every environment (dev, sys, qa, prod).

Last but not least it has an effect on application security. Attackers might be able to decompile the code and thereby discover a potentially sensitive address. They can perform a Denial of Service attack on the service, try to get access to the system, or try to spoof the IP address to bypass security checks. Such attacks can always be possible, but in the case of a hardcoded IP address solving the issue will take more time, which will increase an attack’s impact.

Ask Yourself Whether

The disclosed IP address is sensitive, e.g.:

- Can give information to an attacker about the network topology.
- It’s a personal (assigned to an identifiable person) IP address.

There is a risk if you answered yes to any of these questions.

Recommended Secure Coding Practices

Don’t hard-code the IP address in the source code, instead make it configurable with environment variables, configuration files, or a similar approach. Alternatively, if confidentially is not required a domain name can be used since it allows to change the destination quickly without having to rebuild the software.

Sensitive Code Example

```
ip = "192.168.12.42"; // Sensitive

const net = require('net');
var client = new net.Socket();
client.connect(80, ip, function() {
  // ...
});
```

Compliant Solution

```
ip = process.env.IP_ADDRESS; // Compliant

const net = require('net');
var client = new net.Socket();
client.connect(80, ip, function() {
  // ...
});
```

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Loopback addresses 127.0.0.0/8 in CIDR notation (from 127.0.0.0 to 127.255.255.255)
- Broadcast address 255.255.255.255
- Non routable address 0.0.0.0
- Strings of the form `2.5.<number>.<number>` as they often match Object Identifiers (OID).
- Addresses in the ranges 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, reserved for documentation purposes by RFC 5737
- Addresses in the 2001:db8::/32 range, reserved for documentation purposes by RFC 3849

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

root_cause

Hardcoding IP addresses is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2006-5901
- CVE-2005-3725

Today's services have an ever-changing architecture due to their scaling and redundancy needs. It is a mistake to think that a service will always have the same IP address. When it does change, the hardcoded IP will have to be modified too. This will have an impact on the product development, delivery, and deployment:

- The developers will have to do a rapid fix every time this happens, instead of having an operation team change a configuration file.
- It misleads to use the same address in every environment (dev, sys, qa, prod).

Last but not least it has an effect on application security. Attackers might be able to decompile the code and thereby discover a potentially sensitive address. They can perform a Denial of Service attack on the service, try to get access to the system, or try to spoof the IP address to bypass security checks. Such attacks can always be possible, but in the case of a hardcoded IP address solving the issue will take more time, which will increase an attack's impact.

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Loopback addresses 127.0.0.0/8 in CIDR notation (from 127.0.0.0 to 127.255.255.255)
- Broadcast address 255.255.255.255
- Non routable address 0.0.0.0
- Strings of the form `2.5.<number>.<number>` as they often match Object Identifiers (OID).
- Addresses in the ranges 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, reserved for documentation purposes by RFC 5737
- Addresses in the 2001:db8::/32 range, reserved for documentation purposes by RFC 3849

Octal values should not be used

Clave: javascript:S1314

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Integer literals starting with a zero are octal rather than decimal values. While using octal values is fully supported, most developers do not have experience with them. They may not recognize octal values as such, mistaking them instead for decimal values.

Additionally, these literals will throw SyntaxError in strict mode. 0-prefixed octal literals have been deprecated since ECMAScript 5 and should not be used in modern JavaScript code.

```
const myNumber = 010; // Noncompliant: Deprecated format
```

Use decimal syntax when possible as it is more readable.

```
const myNumber = 8;
```

If octal notation is required, use the standard syntax: a leading zero followed by a lowercase or uppercase Latin letter "O" (`0o` or `0O`).

```
const myNumber = 0o10;
```

resources

Documentation

- MDN web docs - Octal numbers
- MDN web docs - SyntaxError: "0"-prefixed octal literals are deprecated
- MDN web docs - Strict mode

"with" statements should not be used

Clave: javascript:S1321

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `with`
- MDN web docs - Strict mode

root_cause

The `with` statement introduces a new scope, where properties of an object can be accessed directly without having to specify the object's name explicitly. However, using it is generally considered a bad practice and is strongly discouraged.

While it might seem convenient at first, it can lead to several issues:

- The `with` statement can make code more ambiguous and harder to read. When reading the code, it becomes unclear where variables or properties are coming from, as they can be from the object in the `with` statement or any of its parent scopes.
- The `with` statement negatively impacts performance. JavaScript engines have a harder time optimizing code with `with` because it adds uncertainty to variable lookups, which can result in slower execution.
- Using `with` can lead to bugs that are difficult to identify and troubleshoot. If a variable is not found in the object within the `with` statement or its parent scopes, JavaScript will create a new global variable instead, potentially leading to unexpected behavior.

As a result of these issues, ECMAScript 5 (ES5) strict mode explicitly forbids the use of the `with` statement. Strict mode was introduced to enhance code safety and maintainability, and it helps to catch potential issues and discourage the use of problematic language features.

```
let x = 'a';

let foo = {
  y: 1
};

with (foo) { // Noncompliant
  y = 4;     // Updates 'foo.y'
  x = 3;     // Does not add a 'foo.x' property; updates the variable 'x' in the outer scope instead
}

console.log(foo.x + " " + x); // Prints: undefined 3
```

Instead of using `with`, it's best to write more explicit code, accessing object properties directly without relying on the with construct.

```
let x = 'a';

let foo = {
  y: 1
};

foo.y = 4;
foo.x = 3;

console.log(foo.x + " " + x); // Prints: 3 a
```

Statements should end with semicolons

Clave: javascript:S1438

Severidad: MINOR

Estado: ⚠️ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

In JavaScript, the semicolon (`;`) is optional as a statement separator, but omitting semicolons can be confusing, and lead to unexpected results because a semicolon is implicitly inserted at the end of each line.

Noncompliant code example

```
function fun() {
  return  // Noncompliant. ';' implicitly inserted at end of line
       5   // Noncompliant. ';' implicitly inserted at end of line
}
print(fun());  // prints "undefined", not "5"
```

Compliant solution

```
function fun() {
  return 5;
}
print(fun());
```

Only "while", "do", "for" and "switch" statements should be labelled

Clave: javascript:S1439

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - label

Related rules

- S1119 - Labels should not be used

root_cause

Labels allow specifying a target statement to jump to using the `break` or `continue` statements. It’s possible to assign a label to any statement or block of statements. However, using it with any statement can create a complex control flow path, making the code harder to understand and maintain.

```
myLabel: if (i % 2 == 0) { // Noncompliant: Labeling an if statement
  if (i == 12) {
    console.log("12");
    break myLabel;
  }
  console.log("Even number, but not 12");
}
```

Instead of using a label with these nested `if` statements, this code block should be refactored.

```
if (i % 2 == 0) { // Compliant
  if (i == 12) {
    console.log("12");
  } else {
    console.log("Even number, but not 12");
  }
}
```

The rule considers that `while`, `do-while`, `for`, and `switch` statements don’t create complex control flow paths, thus these statements are not reported.

```
outerLoop: for (let i = 0; i < 10; i++) { // Compliant
  for (let j = 0; j < 10; j++) {
    if (condition(i, j)) {
      break outerLoop;
    }
  }
}
```

"===" and "!==" should be used instead of "==" and "!="

Clave: javascript:S1440

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, there are two types of comparison operators: strict and non-strict.

- Strict operators: These operators compare both value and type. They are represented as `===` (strict equality) and `!==` (strict inequality). For example, `5 === "5"` would return `false` because, although the values are the same, the types are different (one is a number, the other is a string).
- Non-Strict operators: These operators compare only value, not type. They are represented as `==` (equality) and `!=` (inequality). For example, `5 == "5"` would return `true` because the values are the same, even though the types are different.

It's generally recommended to use strict operators in JavaScript to avoid unexpected results due to JavaScript's type coercion. This is because non-strict operators can lead to some counter-intuitive results. For example, `0 == false` would return `true`, which might not be the expected outcome.

```
function checkEqual(a, b) {
  if (a == b) { // Noncompliant: using non-strict equality '=='
    return "Equal";
  } else {
    return "Not equal";
  }
}

console.log(checkEqual(0, false)); // Output: "Equal"
```

You should use the strict equality and inequality operators to prevent type coercion, avoid unexpected outcomes when comparing values of different types, and provide more predictable results.

```
function checkEqual(a, b) {
  if (a === b) {
    return "Equal";
  } else {
    return "Not equal";
  }
}

console.log(checkEqual(0, false)); // Output: "Not equal
```

Exceptions

The rule does not report on these cases:

- Comparing two literal values
- Evaluating the value of `typeof`
- Comparing against `null`

resources

Documentation

- MDN web docs - Strict equality
- MDN web docs - Strict inequality
- MDN web docs - Equality
- MDN web docs - Inequality
- MDN web docs - Type coercion
- MDN web docs - Truthy
- MDN web docs - Falsy

Quotes for string literals should be used consistently

Clave: javascript:S1441

Severidad: MAJOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule checks that all string literals use the same kind of quotes.

Noncompliant code example

Using the parameter default (forcing single quotes):

```
var firstParameter = "something"; // Noncompliant
```

Compliant solution

```
var firstParameter = 'something';
```

Exceptions

Strings that contain quotes are ignored.

```
let heSaid = "Then he said 'What?'."  // ignored
let sheSaid = '"Whatever!" she replied.'  // ignored
```

introduction

This rule is deprecated, and will eventually be removed.

"alert(...)" should not be used

Clave: javascript:S1442

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

resources

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code

root_cause

`alert(...)` as well as `confirm(...)` and `prompt(...)` can be useful for debugging during development, but in production mode this kind of pop-up could expose sensitive information to attackers, and should never be displayed.

Noncompliant code example

```
if(unexpectedCondition) {
  alert("Unexpected Condition");
}
```

introduction

This rule is deprecated; use S4507 instead.

Track lack of copyright and license headers

Clave: javascript:S1451

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Each source file should start with a header stating file ownership and the license which must be used to distribute the application.

This rule must be fed with the header text that is expected at the beginning of every file.

Compliant solution

```
/*
 * SonarQube, open source software quality management tool.
 * Copyright (C) 2008-2013 SonarSource
 * mailto:contact AT sonarsource DOT com
 *
 * SonarQube is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 3 of the License, or (at your option) any later version.
 *
 * SonarQube is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
 */
```

Functions should not be defined inside loops

Clave: javascript:S1515

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Defining functions inside loops in JavaScript can lead to several issues and is generally considered bad practice. The main problems associated with this approach are related to performance, scope, and potential unintended behavior:

- When a function is defined inside a loop, the function is re-created on each iteration of the loop. This can cause unnecessary overhead and lead to performance issues, especially if the loop runs repeatedly. Defining the function outside the loop is more efficient so that it is created only once.
- Functions defined inside loops have access to the loop’s variables and parameters. This can sometimes lead to unintended behavior or bugs due to closures. Closures in JavaScript capture the environment in which they are created, including variables and parameters, and this can cause unexpected results when those variables change during the loop.
- Code that defines functions inside loops can be harder to read and maintain, especially for other developers who might not expect functions to be redefined within the loop. Keeping functions separate and clearly defined is better, improving code organization and understandability.

```
for (let i = 0; i < 5; i++) {
  function processItem(item) { // Noncompliant: Function 'processItem' inside a for loop
    // Some processing logic
    console.log(item);
  }

  processItem(i);
}
```

Define the function outside the loop and use the function parameters to pass any needed variables instead.

```
function processItem(item) {
  // Some processing logic
  console.log(item);
}

for (let i = 0; i < 5; i++) {
  processItem(i);
}
```

resources

Documentation

- MDN web docs - Closures
- MDN web docs - Scope

Multiline string literals should not be used

Clave: javascript:S1516

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - String operators

root_cause

Creating multiline strings by using a backslash (\) before a newline is known as "line continuation" or "line breaking." While it may seem like a convenient way to format multiline strings, it is generally considered bad practice.

- Line continuation can make the code harder to read and understand, especially when dealing with long strings. It introduces an extra character at the end of each line, which can clutter the code and reduce its readability.
- If the string content changes, it might require reformatting the entire multiline string, involving adjusting the line breaks and ensuring the backslashes are correctly placed. This can be error-prone and cumbersome, leading to maintenance issues.
- Line continuation can sometimes behave unexpectedly, particularly when there are trailing spaces or tabs after the backslash. This can lead to subtle bugs that are difficult to spot and debug.

```
let myString = 'A rather long string of English text, an error message \
                actually that just keeps going and going -- an error \
                message to make the Energizer bunny blush (right through \
                those Schwarzenegger shades)! Where was I? Oh yes, \
                you\'ve got an error and all the extraneous whitespace is \
                just gravy.  Have a nice day.';  // Noncompliant
```

Instead, you should use string concatenation for multiline strings, which involves combining multiple strings to create a single string that spans multiple lines.

```
let myString = 'A rather long string of English text, an error message ' +
               'actually that just keeps going and going -- an error ' +
               'message to make the Energizer bunny blush (right through ' +
               'those Schwarzenegger shades)! Where was I? Oh yes, ' +
               'you\'ve got an error and all the extraneous whitespace is ' +
               'just gravy.  Have a nice day.';
```

Dynamically executing code is security-sensitive

Clave: javascript:S1523

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

default

Executing code dynamically is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-9807
- CVE-2017-9802

Some APIs enable the execution of dynamic code by providing it as strings at runtime. These APIs might be useful in some very specific meta-programming use-cases. However most of the time their use is frowned upon as they also increase the risk of Injected Code. Such attacks can either run on the server or in the client (exemple: XSS attack) and have a huge impact on an application’s security.

This rule raises issues on calls to `eval` and `Function` constructor. This rule does not detect code injections. It only highlights the use of APIs which should be used sparingly and very carefully. The goal is to guide security code reviews.

The rule also flags string literals starting with `javascript:` as the code passed in `javascript:` URLs is evaluated the same way as calls to `eval` or `Function` constructor.

Ask Yourself Whether

- the executed code may come from an untrusted source and hasn’t been sanitized.
- you really need to run code dynamically.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Regarding the execution of unknown code, the best solution is to not run code provided by an untrusted source. If you really need to do it, run the code in a sandboxed environment. Use jails, firewalls and whatever means your operating system and programming language provide (example: Security Managers in java, iframes and same-origin policy for javascript in a web browser).

Do not try to create a blacklist of dangerous code. It is impossible to cover all attacks that way.

Avoid using dynamic code APIs whenever possible. Hard-coded code is always safer.

Sensitive Code Example

```
let value = eval('obj.' + propName); // Sensitive
let func = Function('obj' + propName); // Sensitive
location.href = 'javascript:void(0)'; // Sensitive
```

Exceptions

This rule will not raise an issue when the argument of the `eval` or `Function` is a literal string as it is reasonably safe.

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-95 - Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

root_cause

Executing code dynamically is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-9807
- CVE-2017-9802

Some APIs enable the execution of dynamic code by providing it as strings at runtime. These APIs might be useful in some very specific meta-programming use-cases. However most of the time their use is frowned upon as they also increase the risk of Injected Code. Such attacks can either run on the server or in the client (exemple: XSS attack) and have a huge impact on an application's security.

This rule raises issues on calls to `eval` and `Function` constructor. This rule does not detect code injections. It only highlights the use of APIs which should be used sparingly and very carefully. The goal is to guide security code reviews.

The rule also flags string literals starting with `javascript:` as the code passed in `javascript:` URLs is evaluated the same way as calls to `eval` or `Function` constructor.

Exceptions

This rule will not raise an issue when the argument of the `eval` or `Function` is a literal string as it is reasonably safe.

how_to_fix

Recommended Secure Coding Practices

Regarding the execution of unknown code, the best solution is to not run code provided by an untrusted source. If you really need to do it, run the code in a sandboxed environment. Use jails, firewalls and whatever means your operating system and programming language provide (example: Security Managers in java, iframes and same-origin policy for javascript in a web browser).

Do not try to create a blacklist of dangerous code. It is impossible to cover all attacks that way.

Avoid using dynamic code APIs whenever possible. Hard-coded code is always safer.

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-95 - Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

assess_the_problem

Ask Yourself Whether

- the executed code may come from an untrusted source and hasn't been sanitized.
- you really need to run code dynamically.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
let value = eval('obj.' + propName); // Sensitive
let func = Function('obj' + propName); // Sensitive
location.href = 'javascript:void(0)'; // Sensitive
```

Debugger statements should not be used

Clave: javascript:S1525

Severidad: MINOR

Estado: ⚠️ Depreciada

Descripción: No disponible

root_cause

The debugger statement can be placed anywhere in procedures to suspend execution. Using the debugger statement is similar to setting a breakpoint in the code. By definition such statement must absolutely be removed from the source code to prevent any unexpected behavior or added vulnerability to attacks in production.

Noncompliant code example

```
for (i = 1; i<5; i++) {
  // Print i to the Output window.
  Debug.write("loop index is " + i);
  // Wait for user to resume.
  debugger;
}
```

Compliant solution

```
for (i = 1; i<5; i++) {
  // Print i to the Output window.
  Debug.write("loop index is " + i);
}
```

resources

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code

introduction

This rule is deprecated; use S4507 instead.

Variables declared with "var" should be declared before they are used

Clave: javascript:S1526

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Variables declared with `var` have the special property that regardless of where they’re declared in a function they "float" to the top of the function and are available for use even before they’re declared. That makes scoping confusing, especially for new coders.

To keep confusion to a minimum, `var` declarations should happen before they are used for the first time.

Noncompliant code example

```
var x = 1;

function fun(){
  alert(x); // Noncompliant as x is declared later in the same scope
  if(something) {
    var x = 42; // Declaration in function scope (not block scope!) shadows global variable
  }
}

fun(); // Unexpectedly alerts "undefined" instead of "1"
```

Compliant solution

```
var x = 1;

function fun() {
  print(x);
  if (something) {
    x = 42;
  }
}

fun(); // Print "1"
```

Future reserved words should not be used as identifiers

Clave: javascript:S1527

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

The ECMAScript specification defines a set of special words as future keywords of the language. They don’t have particular meaning for now, but they might at some future time.

The list contains the following words:

- `await`
- `class`
- `const`
- `enum`
- `export`
- `extends`
- `implements`
- `import`
- `interface`
- `let`
- `package`
- `private`
- `protected`
- `public`
- `static`
- `super`
- `yield`

Some of these words have already been adopted by current versions of ECMAScript, but they are kept to consider legacy JavaScript codebases as well. Others are only reserved when used in strict mode.

These future reserved words should be avoided because they may cause syntax errors if they are ever adopted.

```
const package = document.getElementsByName("foo"); // Noncompliant: `package` is used as an identifier here
```

These future keywords can be used anywhere if it is not identifiers.

```
const someData = { package: true };                // Compliant: `package` is not used as an identifier here
```

resources

Documentation

- MDN web docs - Future reserved words
- MDN web docs - Strict mode
- Wikipedia - Syntax error

Array constructors should not be used

Clave: javascript:S1528

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Array literals should always be preferred to Array constructors.

Array constructors are error-prone due to the way their arguments are interpreted. If more than one argument is used, the array length will be equal to the number of arguments. However, using a single argument will have one of three consequences:

- If the argument is a number and it is a natural number the length will be equal to the value of the argument.

```
let arr = new Array(3); // [empty × 3]
```

- If the argument is a number, but not a natural number an exception will be thrown.

```
let arr = new Array(3.14);  // RangeError: Invalid array length
```

- Otherwise the array will have one element with the argument as its value.

```
let arr = new Array("3");  // ["3"]
```

Note that even if you set the length of an array, it will be empty. That is, it will have the number of elements you declared, but they won’t contain anything, so no callbacks will be applied to the array elements.

For these reasons, if someone changes the code to pass 1 argument instead of 2 arguments, the array might not have the expected length. To avoid these kinds of weird cases, always use the more readable array literal initialization format.

Noncompliant code example

```
let myArray = new Array(x1, x2, x3);   // Noncompliant. Results in 3-element array.
let emptyArray = new Array();          // Noncompliant. Results in 0-element array.

let unstableArray = new Array(n);      // Noncompliant. Variable in results.

let arr = new Array(3); // Noncompliant; empty array of length 3
arr.foreach((x) => alert("Hello " + x)); // callback is not executed because there's nothing in arr
let anotherArr = arr.map(() => 42); // anotherArr is also empty because callback didn't execute
```

Compliant solution

```
let myArray = [x1, x2, x3];
let emptyArray = [];

// if "n" is the only array element
let unstableArray = [n];
// or,  if "n" is the array length (since ES 2015)
let unstableArray = Array.from({length: n});

let arr = ["Elena", "Mike", "Sarah"];
arr.foreach((x) => alert("Hello " + x));
let anotherArr = arr.map(() => 42);  // anotherArr now holds 42 in each element
```

Bitwise operators should not be used in boolean contexts

Clave: javascript:S1529

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Bitwise AND (`&`)
- MDN web docs - Bitwise OR (`|`)
- MDN web docs - Logical AND (`&&`)
- MDN web docs - Logical OR (`||`)
- MDN web docs - Truthy
- MDN web docs - Falsy

root_cause

Bitwise operations are operations that manipulate individual bits in binary representations of numbers. These operations work at the binary level, treating numbers as sequences of 32 bits (in 32-bit environments) or 64 bits (in 64-bit environments). However, they should not be used in a boolean context because they have different behaviors compared to logical operators when applied to boolean values:

- When applied to boolean values, bitwise AND (`&`) and OR (`|`) perform bitwise operations on the binary representation of the numbers. They treat the operands as 32-bit signed integers and manipulate their individual bits.
- Logical AND (`&&`) and OR (`||`) are specifically designed for boolean operations. They return a boolean value based on the truthiness or falsiness of the operands.`&&` returns `true` if both operands are truthy; otherwise, it returns `false`. `||` operator returns `true` if at least one of the operands is truthy; otherwise, it returns `false`.

Bitwise operators & and | can be easily mistaken for logical operators && and ||, especially for those who are not familiar with the distinction between them or their specific use cases.

This rule raises an issue when & or | is used in a boolean context.

```
if (a & b) { /* ... */ } // Noncompliant: The operator & is used in a boolean context
```

You should use the logical variant of the bitwise operator, that is, && instead of & and || instead of |.

```
if (a && b) { /* ... */ }
```

Exceptions

When a file contains other bitwise operations, (^, <<, >>>, >>, ~, &=, ^=, |=, <<=, >>=, >>>=, and & or | used with a numeric literal as the right operand) all issues in the file are ignored, because it is evidence that bitwise operations were truly intended.

Function declarations should not be made within blocks

Clave: javascript:S1530

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

While most script engines support function declarations within blocks, from browser to browser, the implementations are inconsistent with each other.

Noncompliant code example

```
if (x) {
  function foo() {} //foo is hoisted in Chrome, Firefox and Safari, but not in Edge.
}
```

Compliant solution

```
if (x) {
  const foo = function() {}
}
```

Wrapper objects should not be used for primitive types

Clave: javascript:S1533

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The JavaScript wrapper objects `Number`, `String`, and `Boolean` provide a way to work with their respective primitive types (`number`, `string` and `boolean`) as objects.

Using wrapper can lead to unexpected behavior due to the differences in how they are compared and used in operations compared to primitive types. It can also lead to unnecessary memory allocation and slower code execution.

```
let x = new Number("0"); // Noncompliant: x is an object, not a primitive
if (x) {
  alert('hi');  // Shows 'hi'.
}
```

Remove the `new` keyword to get the primitive value instead of a wrapper object.

```
let x = Number("0");
if (x) {
  alert('hi');
}
```

However, it is generally recommended to use primitive types directly instead of wrapper objects, which makes the code more consistent and easier to understand.

```
let x = 0;
if (x) {
  alert('hi');
}
```

resources

Documentation

- MDN web docs - Primitive
- MDN web docs - `Number()` constructor
- MDN web docs - `String()` constructor
- MDN web docs - `Boolean()` constructor

Member names should not be duplicated within a class or object literal

Clave: javascript:S1534

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

JavaScript allows duplicate property names in classes and object literals. The last occurrence will overwrite previous definitions. Therefore, having more than one occurrence will have no effect and may cause misunderstandings and bugs.

```
let data = {
  "key": "value",
  "1": "value",
  "key": "value", // Noncompliant - duplicate of "key"
  'key': "value", // Noncompliant - duplicate of "key"
  key: "value", // Noncompliant - duplicate of "key"
  \u006bey: "value", // Noncompliant - duplicate of "key"
  "\u006bey": "value", // Noncompliant - duplicate of "key"
  "\x6bey": "value", // Noncompliant - duplicate of "key"
  1: "value" // Noncompliant - duplicate of "1"
}

class Foo {
  bar() { }
  bar() { }  // Noncompliant - duplicate of "bar"
}
```

Defining a `class` with a duplicated `constructor` will generate an error.

```
class Class {
  constructor() {
  }
  constructor(value) { // Noncompliant: A class may only have one constructor
  }
}
```

Before ECMAScript 2015, using duplicate names generated an error in JavaScript strict mode code.

This rule will also report on duplicate properties in JSX.

```
function MyComponent(props) {
  return <div prop={props.prop1} prop={props.prop2}> { /* Noncompliant, 'prop' is defined twice */ }
    This is my component
  </div>;
}
```

resources

Documentation

- MDN web docs - Property definitions

"for...in" loops should filter properties before acting on them

Clave: javascript:S1535

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `for...in` statement allows you to loop through the names of all of the properties of an object. The list of properties includes all those properties that were inherited through the prototype chain. This has the side effect of serving up functions when the interest is in data properties. Programs that don't take this into account can fail.

Therefore, the body of every `for...in` statement should be wrapped in an `if` statement that filters which properties are acted upon. It can select for a particular type or range of values, or it can exclude functions, or it can exclude properties from the prototype.

Noncompliant code example

```
for (name in object) {
    doSomething(name);  // Noncompliant
}
```

Compliant solution

```
for (name in object) {
  if (object.hasOwnProperty(name)) {
    doSomething(name);
  }
}
```

Exceptions

Loops used to clone objects are ignored.

```
for (prop in obj) {
  a[prop] = obj[prop];  // Compliant by exception
}
```

Function argument names should be unique

Clave: javascript:S1536

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Functions
- MDN web docs - The arguments object
- MDN web docs - Strict mode

root_cause

Function parameter names should be unique in JavaScript. Unique parameter names ensure that there is no ambiguity in referring to specific parameters within the function body. If multiple parameters share the same name, it becomes unclear which parameter is being referred to when using that name within the function.

Unique parameter names improve the readability and maintainability of code. When parameter names are descriptive and distinct, it becomes easier for other developers (including yourself) to understand the purpose and functionality of the function.

When parameter names are not unique, the later occurrence of a parameter will overwrite the earlier occurrence, potentially leading to unintended consequences or bugs. This behavior can cause confusion and make the code harder to debug.

```
function f(a, b, a) { // Noncompliant: The first occurrence of `a` will be overwritten by the later occurrence
  console.log(a, b);
}

f(1, 2, 3);           // Outputs 5
```

In strict mode, JavaScript enforces stricter rules and detects potential issues. Duplicate parameter names are considered a syntax error in strict mode. By using unique parameter names, you ensure compatibility with strict mode and can benefit from the enhanced error checking and code quality improvements it provides.

```
'use strict';

function f(a, b, a) { // Noncompliant: SyntaxError: Duplicate parameter name not allowed in this context
  console.log(a, b);
}

f(1, 2, 3);
```

You should remove the duplicates or rename them while carefully ensuring you are not altering the semantics of your code.

```
function f(a, b, c) {
  console.log(a, b, c);
}

f(1, 2, 3);
```

Trailing commas should not be used

Clave: javascript:S1537

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

Most browsers parse and discard a meaningless, trailing comma. Unfortunately, that’s not the case for Internet Explorer below version 9, which throws a meaningless error. Therefore trailing commas should be eliminated.

Noncompliant code example

```
var settings = {
    'foo'  : oof,
    'bar' : rab,    // Noncompliant - trailing comma
};
```

Compliant solution

```
var settings = {
    'foo'  : oof,
    'bar' : rab
};
```

introduction

This rule is deprecated, and will eventually be removed.

"strict" mode should be used with caution

Clave: javascript:S1539

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Even though it may be a good practice to enforce JavaScript strict mode, doing so could result in unexpected behaviors on browsers that do not support it yet. Using this feature should therefore be done with caution and with full knowledge of the potential consequences on browsers that do not support it.

Noncompliant code example

```
function strict() {
  'use strict';
}
```

Cyclomatic Complexity of functions should not be too high

Clave: javascript:S1541

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

The Cyclomatic Complexity of functions should not exceed a defined threshold. Complex code may perform poorly and can be difficult to test thoroughly.

Variables should not be self-assigned

Clave: javascript:S1656

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

There is no reason to re-assign a variable to itself. Either this statement is redundant and should be removed, or the re-assignment is a mistake and some other value or variable was intended for the assignment instead.

Noncompliant code example

```
function setName(name) {
    name = name;
}
```

Compliant solution

```
function setName(name) {
    this.name = name;
}
```

Loops with at most one iteration should be refactored

Clave: javascript:S1751

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A loop with at most one iteration is equivalent to the use of an `if` statement to conditionally execute one piece of code. No developer expects to find such a use of a loop statement. If the initial intention of the author was really to conditionally execute one piece of code, an `if` statement should be used instead.

At worst that was not the initial intention of the author and so the body of the loop should be fixed to use the nested `return`, `break` or `throw` statements in a more appropriate way.

Noncompliant code example

```
for (let i = 0; i < 10; i++) { // noncompliant, loop only executes once
  console.log("i is " + i);
  break;
}
...
for (let i = 0; i < 10; i++) { // noncompliant, loop only executes once
  if (i == x) {
    break;
  } else {
    console.log("i is " + i);
    return;
  }
}
```

Compliant solution

```
for (let i = 0; i < 10; i++) {
  console.log("i is " + i);
}
...
for (let i = 0; i < 10; i++) {
  if (i == x) {
    break;
  } else {
    console.log("i is " + i);
  }
}
```

All code should be reachable

Clave: javascript:S1763

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `return`
- MDN web docs - `break`
- MDN web docs - `throw`
- MDN web docs - `continue`
- MDN web docs - Warning: unreachable code after return statement

root_cause

Unreachable code is the code whose statements cannot be executed under any circumstances. Jump statements, like `return`, `break`, `continue`, and `throw`, alter the normal flow of control within a program, making it possible to skip certain parts of the code, terminate loops prematurely, or exit from functions. So any statements that come after a jump are effectively unreachable.

Unreachable statements can be a sign of a logical error or oversight in the program’s design, leading to unexpected behavior at runtime.

```
function func(a) {
  let i = 10;
  return i + a;
  i++; // Noncompliant: this is never executed
}
```

Identify and remove unreachable statements from your code.

```
function func(a) {
  let i = 10;
  return i + a;
}
```

Identical expressions should not be used on both sides of a binary operator

Clave: javascript:S1764

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using the same value on both sides of a binary operator is a code defect. In the case of logical operators, it is either a copy/paste error and, therefore, a bug, or it is simply duplicated code and should be simplified. In the case of bitwise operators and most binary mathematical operators, having the same value on both sides of an operator yields predictable results and should be simplified as well.

Noncompliant code example

```
if (a == b && a == b) { // if the first one is true, the second one is too
  doX();
}
if (a > a) { // always false
  doW();
}

var j = 5 / 5; //always 1
var k = 5 - 5; //always 0
```

Exceptions

The specific case of testing one variable against itself is a valid test for `NaN` and is therefore ignored.

Similarly, left-shifting 1 onto 1 is common in the construction of bit masks, and is ignored.

Moreover comma operator `,` and `instanceof` operator are ignored as there are use-cases when there usage is valid.

```
if (f !== f) { // test for NaN value
  console.log("f is NaN");
}

var i = 1 << 1; // Compliant
var j = a << a; // Noncompliant
```

resources

- S1656 - Implements a check on `=`.

The ternary operator should not be used

Clave: javascript:S1774

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Ternary expressions, while concise, can often lead to code that is difficult to read and understand, especially when they are nested or complex. Prioritizing readability fosters maintainability and reduces the likelihood of bugs. Therefore, they should be removed in favor of more explicit control structures, such as `if`/`else` statements, to improve the clarity and readability of the code.

Noncompliant code example

```
function foo(a) {
  var b = (a === 'A') ? 'is A' : 'is not A'; // Noncompliant
  // ...
}
```

Compliant solution

```
function foo(a) {
  var b;
  if (a === 'A') {
    b = 'is A';
  }
  else {
    b = 'is not A';
  }
  // ...
}
```

Function parameters with default values should be last

Clave: javascript:S1788

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Default parameter values allow callers to specify as many or as few arguments as they want while getting the same functionality and minimizing boilerplate, wrapper code, making a function easier to use.

All function parameters with default values should be declared after the function parameters without default values. Otherwise, it makes it impossible for callers to take advantage of defaults; they must re-specify the defaulted values or pass `undefined` to be able to specify the non-default parameters.

```
function multiply(a = 1, b) { // Noncompliant: parameter with default value should be last
  return a*b;
}

let x = multiply(1, 42); // Cannot benefit from default value
```

Reorder the function parameters so that the ones with default values come after the ones without default values.

```
function multiply(b, a = 1) {
  return a*b;
}

let x = multiply(42);
```

Exceptions

When writing Redux reducers, there is a convention to use default argument syntax to provide initial state (first argument), while action (second argument) is mandatory. A reducer may be called with `undefined` as the state value when the application is being initialized.

```
// Use the initialState as a default value
export default function appReducer(state = initialState, action) {
  switch (action.type) {
    default:
      return state;
  }
}
```

resources

Documentation

- MDN web docs - Default parameters
- Redux Documentation - Writing Reducers

"switch" statements should not be nested

Clave: javascript:S1821

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Nested `switch` structures are difficult to understand because you can easily confuse the cases of an inner `switch` as belonging to an outer statement. Therefore nested `switch` statements should be avoided.

Specifically, you should structure your code to avoid the need for nested `switch` statements, but if you cannot, then consider moving the inner `switch` to another function.

Noncompliant code example

```
function foo(n, m) {
  switch (n) {
    case 0:
      switch (m) {  // Noncompliant; nested switch
        // ...
      }
    case 1:
      // ...
    default:
      // ...
  }
}
```

Compliant solution

```
function foo(n, m) {
  switch (n) {
    case 0:
      bar(m);
    case 1:
      // ...
    default:
      // ...
  }
}

function bar(m) {
  switch(m) {
    // ...
  }
}
```

Two branches in a conditional structure should not have exactly the same implementation

Clave: javascript:S1871

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Related rules

- S3923 - All branches in a conditional structure should not have exactly the same implementation

root_cause

When the same code is duplicated in two or more separate branches of a conditional, it can make the code harder to understand, maintain, and can potentially introduce bugs if one instance of the code is changed but others are not.

Having two `cases` in a `switch` statement or two branches in an `if` chain with the same implementation is at best duplicate code, and at worst a coding error.

```
if (a >= 0 && a < 10) {
  doFirstThing();
  doTheThing();
}
else if (a >= 10 && a < 20) {
  doTheOtherThing();
}
else if (a >= 20 && a < 50) {
  doFirstThing();
  doTheThing();  // Noncompliant; duplicates first condition
}
else {
  doTheRest();
}
```

```
switch (i) {
  case 1:
    doFirstThing();
    doSomething();
    break;
  case 2:
    doSomethingDifferent();
    break;
  case 3:  // Noncompliant; duplicates case 1's implementation
    doFirstThing();
    doSomething();
    break;
  default:
    doTheRest();
}
```

If the same logic is truly needed for both instances, then:

- in an `if` chain they should be combined

```
if ((a >= 0 && a < 10) || (a >= 20 && a < 50)) { // Compliant
  doFirstThing();
  doTheThing();
}
else if (a >= 10 && a < 20) {
  doTheOtherThing();
}
else {
  doTheRest();
}
```

- for a `switch`, one should fall through to the other

```
switch (i) {
  case 1:
  case 3: // Compliant
    doFirstThing();
    doSomething();
    break;
  case 2:
    doSomethingDifferent();
    break;
  default:
    doTheRest();
}
```

When all blocks are identical, either this rule will trigger if there is no default clause or rule S3923 will raise if there is a default clause.

Exceptions

Unless all blocks are identical, blocks in an `if` chain that contain a single line of code are ignored. The same applies to blocks in a `switch` statement that contains a single line of code with or without a following `break`.

```
if (a == 1) {
  doSomething();  // Compliant, usually this is done on purpose to increase the readability
} else if (a == 2) {
  doSomethingElse();
} else {
  doSomething();
}
```

Deprecated APIs should not be used

Clave: javascript:S1874

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Code is sometimes annotated as deprecated by developers maintaining libraries or APIs to indicate that the method, class, or other programming element is no longer recommended for use. This is typically due to the introduction of a newer or more effective alternative. For example, when a better solution has been identified, or when the existing code presents potential errors or security risks.

Deprecation is a good practice because it helps to phase out obsolete code in a controlled manner, without breaking existing software that may still depend on it. It is a way to warn other developers not to use the deprecated element in new code, and to replace it in existing code when possible.

Deprecated classes, interfaces, and their members should not be used, inherited or extended because they will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology.

Check the documentation or the deprecation message to understand why the code was deprecated and what the recommended alternative is.

```
/**
 * @deprecated Use newFunction instead.
 */
function oldFunction() {
  console.log("This is the old function.");
}

function newFunction() {
  console.log("This is the new function.");
}
oldFunction(); // Noncompliant: "oldFunction is deprecated"
```

resources

Documentation

- CWE - CWE-477 - Use of Obsolete Functions

Boolean checks should not be inverted

Clave: javascript:S1940

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

It is needlessly complex to invert the result of a boolean comparison. The opposite comparison should be made instead.

Noncompliant code example

```
if (!(a === 2)) { ... }  // Noncompliant
```

Compliant solution

```
if (a !== 2) { ... }
```

"for" loop increment clauses should modify the loops' counters

Clave: javascript:S1994

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

It can be extremely confusing when a `for` loop's counter is incremented outside of its increment clause. In such cases, the increment should be moved to the loop's increment clause if at all possible.

Noncompliant code example

```
for (i = 0; i < 10; j++) { // Noncompliant
  // ...
  i++;
}
```

Compliant solution

```
for (i = 0; i < 10; i++, j++) {
  // ...
}
```

Or

```
for (i = 0; i < 10; i++) {
  // ...
  j++;
}
```

Special identifiers should not be bound or assigned

Clave: javascript:S2137

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Reserved words
- MDN web docs - Identifiers with special meanings
- MDN web docs - Global Objects

root_cause

JavaScript has special identifiers that, while not reserved, still should not be used as identifiers. They form the JavaScript standard built-in objects and global properties. They are available in all environments. Some examples are:

- Global objects: `Object`, `Function`, `Error`, …
- Global object function properties: `eval()`, `isNan()`, `parseFloat()`, `decodeURI()`, …
- Global object value properties: `undefined`, `NaN`, `Infinity`
- Identifiers with special meanings: `arguments`

These words should not be bound or assigned, because doing so would overwrite the original definitions of these identifiers. What’s more, assigning or binding some of these names will generate an error in JavaScript strict mode code.

Noncompliant code example

```
eval = 17; // Noncompliant
arguments++; // Noncompliant
++eval; // Noncompliant
const obj = { set p(arguments) { } }; // Noncompliant
let eval; // Noncompliant
try { /* ... */ } catch (arguments) { } // Noncompliant
function x(eval) { /* ... */ } // Noncompliant
function arguments() { /* ... */ } // Noncompliant
const y = function eval() { /* ... */ }; // Noncompliant

function fun() {
  if (arguments.length == 0) { // Compliant
    // do something
  }
}
```

Compliant solution

```
result = 17;
args++;
++result;
const obj = { set p(arg) { } };
let result;
try { /* ... */ } catch (args) { }
function x(arg) { /* ... */ }
function args() { /* ... */ }
const y = function fun() { /* ... */ };

function fun() {
  if (arguments.length == 0) {
    // do something
  }
}
```

"undefined" should not be assigned

Clave: javascript:S2138

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

`undefined` is the value you get for variables and properties which have not yet been created. Use the same value to reset an existing variable and you lose the ability to distinguish between a variable that exists but has no value and a variable that does not yet exist. Instead, `null` should be used, allowing you to tell the difference between a property that has been reset and one that was never created.

Noncompliant code example

```
var myObject = {};

// ...
myObject.fname = undefined;  // Noncompliant
// ...

if (myObject.lname == undefined) {
  // property not yet created
}
if (myObject.fname == undefined) {
  // no real way of knowing the true state of myObject.fname
}
```

Compliant solution

```
var myObject = {};

// ...
myObject.fname = null;
// ...

if (myObject.lname == undefined) {
  // property not yet created
}
if (myObject.fname == undefined) {
  // no real way of knowing the true state of myObject.fname
}
```

A "for" loop update clause should move the counter in the right direction

Clave: javascript:S2251

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In a `for` loop, the update clause is responsible for modifying the loop counter variable in the appropriate direction to control the loop’s iteration. It determines how the loop counter variable changes with each iteration of the loop. The loop counter should move in the right direction to prevent infinite loops or unexpected behavior.

This rule raises an issue when the loop counter is updated in the wrong direction with respect to the loop termination condition.

```
for (let i = 0; i < strings.length; i--) { // Noncompliant: The counter 'i' is decremented, making the loop infinite
  //...
}
```

To ensure the `for` loop behaves as expected, you should specify the correct update clause that moves the loop counter in the right direction based on the loop’s logic and desired outcome.

```
for (let i = 0; i < strings.length; i++) {
  //...
}
```

resources

Documentation

- MDN web docs - `for`
- Wikipedia - Infinite loop

Writing cookies is security-sensitive

Clave: javascript:S2255

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule is deprecated, and will eventually be removed.

Using cookies is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-11639
- CVE-2016-6537

Attackers can use widely-available tools to read cookies. Any sensitive information they may contain will be exposed.

This rule flags code that writes cookies.

assess_the_problem

Ask Yourself Whether

- sensitive information is stored inside the cookie.

You are at risk if you answered yes to this question.

Sensitive Code Example

```
// === Built-in NodeJS modules ===
const http = require('http');
const https = require('https');

http.createServer(function(req, res) {
  res.setHeader('Set-Cookie', ['type=ninja', 'lang=js']); // Sensitive
});
https.createServer(function(req, res) {
  res.setHeader('Set-Cookie', ['type=ninja', 'lang=js']); // Sensitive
});

// === ExpressJS ===
const express = require('express');
const app = express();
app.use(function(req, res, next) {
  res.cookie('name', 'John'); // Sensitive
});

// === In browser ===
// Set cookie
document.cookie = "name=John"; // Sensitive
```

how_to_fix

Recommended Secure Coding Practices

Cookies should only be used to manage the user session. The best practice is to keep all user-related information server-side and link them to the user session, never sending them to the client. In a very few corner cases, cookies can be used for non-sensitive information that need to live longer than the user session.

Do not try to encode sensitive information in a non human-readable format before writing them in a cookie. The encoding can be reverted and the original information will be exposed.

Using cookies only for session IDs doesn't make them secure. Follow OWASP best practices when you configure your cookies.

As a side note, every information read from a cookie should be Sanitized.

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-312 - Cleartext Storage of Sensitive Information
- CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie
- Derived from FindSecBugs rule COOKIE_USAGE

default

This rule is deprecated, and will eventually be removed.

Using cookies is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-11639
- CVE-2016-6537

Attackers can use widely-available tools to read cookies. Any sensitive information they may contain will be exposed.

This rule flags code that writes cookies.

Ask Yourself Whether

- sensitive information is stored inside the cookie.

You are at risk if you answered yes to this question.

Recommended Secure Coding Practices

Cookies should only be used to manage the user session. The best practice is to keep all user-related information server-side and link them to the user session, never sending them to the client. In a very few corner cases, cookies can be used for non-sensitive information that need to live longer than the user session.

Do not try to encode sensitive information in a non human-readable format before writing them in a cookie. The encoding can be reverted and the original information will be exposed.

Using cookies only for session IDs doesn't make them secure. Follow OWASP best practices when you configure your cookies.

As a side note, every information read from a cookie should be Sanitized.

Sensitive Code Example

```
// === Built-in NodeJS modules ===
const http = require('http');
const https = require('https');

http.createServer(function(req, res) {
  res.setHeader('Set-Cookie', ['type=ninja', 'lang=js']); // Sensitive
});
https.createServer(function(req, res) {
  res.setHeader('Set-Cookie', ['type=ninja', 'lang=js']); // Sensitive
});

// === ExpressJS ===
const express = require('express');
const app = express();
app.use(function(req, res, next) {
  res.cookie('name', 'John'); // Sensitive
});

// === In browser ===
// Set cookie
document.cookie = "name=John"; // Sensitive
```

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-312 - Cleartext Storage of Sensitive Information
- CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie
- Derived from FindSecBugs rule COOKIE_USAGE

Properties of variables with "null" or "undefined" values should not be accessed

Clave: javascript:S2259

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Noncompliant code example

```
let foo = null;
console.log(foo.bar); // Noncompliant: TypeError will be thrown
```

Compliant solution

The simplest solution is to check in a `if` condition the equality to `null` or `undefined`. With the abstract equality operator it is not required to check both, as these operators consider `null` and `undefined` to be equals.

```
let foo;
if (foo != null) {
  console.log(foo.bar);
}
```

Also, the logical AND operator (&&) can be used to check if a variable is truthy before attempting to access its properties. The expression will short-circuit and return the falsy value instead of attempting to access its properties.

```
let foo = null;
console.log(foo && foo.bar);
```

Alternatively, use the optional chaining operator (`?.`) to check if the variable is `null` or `undefined` before attempting to access its property. This operator is more readable and concise, especially when dealing with nested properties.

```
let foo = null;
console.log(foo?.bar);
```

root_cause

In JavaScript, `null` and `undefined` are primitive values that do not have properties or methods. When accessing a property on a `null` or `undefined` value, JavaScript tries to access the property of an object that does not exist, which results in a `TypeError`.

This can cause the program to crash or behave unexpectedly, which can be difficult to debug.

resources

Documentation

- MDN web docs - `undefined`
- MDN web docs - `null`
- MDN web docs - `TypeError`
- MDN web docs - TypeError: "x" has no properties
- MDN web docs - Optional chaining (`?.`)
- MDN web docs - Logical AND (`&&`)

Methods should not contain selector parameters

Clave: javascript:S2301

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A selector parameter is a `boolean` parameter that’s used to determine which of two paths to take through a method. Specifying such a parameter may seem innocuous, particularly if it’s well named.

Unfortunately, developers calling the method won’t see the parameter name, only its value. They’ll be forced either to guess at the meaning or to take extra time to look the method up.

This rule finds methods with a `boolean` that’s used to determine which path to take through the method.

Noncompliant code example

```
function tempt(name: string, ofAge: boolean) {
  if (ofAge) {
    offerLiquor(name);
  } else {
    offerCandy(name);
  }
}

// ...
function corrupt() {
  tempt("Joe", false); // does this mean not to temp Joe?
}
```

Compliant solution

Instead, separate methods should be written.

```
function temptAdult(name: string) {
  offerLiquor(name);
}

function temptChild(name: string) {
  offerCandy(name);
}

// ...
function corrupt() {
  age < legalAge ? temptChild("Joe") : temptAdult("Joe");
}
```

Loop counters should not be assigned within the loop body

Clave: javascript:S2310

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - for
- MDN web docs - `for...of`
- MDN web docs - Iteration protocols

root_cause

Loop counters, such as variables used to track the iteration count in loops, should not be assigned from within the loop body to avoid unexpected behavior and bugs. It can inadvertently lead to an infinite loop or make the loop behavior more complex and harder to reason about.

```
const names = [ "Jack", "Jim", "", "John" ];
for (let i = 0; i < names.length; i++) {
  if (!names[i]) {
    i = names.length; // Noncompliant: The loop counter i is assigned within the loop body
  } else {
    console.log(names[i]);
  }
}
```

To avoid these issues, you should update the loop counter only in the loop's update statement, typically located at the end of the loop body or within the loop header.

```
const names = [ "Jack", "Jim", "", "John" ];
for (let i = 0; i < names.length; i++) {
  if (!names[i]) {
    break;
  } else {
    console.log(names[i]);
  }
}
```

Alternatively, you should use the `for…of` statement if your intention is only to iterate over the values of an iterable object.

```
const names = [ "Jack", "Jim", "", "John" ];
for (const name of names) {
  if (!name) {
    break;
  } else {
    console.log(name);
  }
}
```

Property getters and setters should come in pairs

Clave: javascript:S2376

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When an object is created with a setter for a property but without a getter for that property, the property is inaccessible and is thus useless.

This rule also enforces the reverse situation (getter but no setter).

Noncompliant code example

```
var obj = {
    set foo(value) {
        this.fooval = value;
    }
};
```

Compliant solution

```
var obj = {
    set foo(value) {
        this.fooval = value;
    },
    get foo() {
        return this.fooval;
    }
};
```

or

```
var obj = {
    setFoo(value) {    // a standard method, not a setter
        this.fooval = value;
    }
};
```

Built-in objects should not be overridden

Clave: javascript:S2424

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Overriding an object changes its behavior and could potentially impact all code using that object. Overriding standard, built-in objects could therefore have broad, potentially catastrophic effects on previously-working code.

This rule detects overrides of the following native objects:

- Fundamental objects - Object, Function, Boolean, Symbol, Error, EvalError, InternalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError
- Numbers and dates - Number, Math, Date
- Text processing - String, RegExp
- Indexed collections - Array, Int8Array, Uint8Array, Uint8ClampedArray, Int16Array, Unit16Array, Int32Array, Uint32Array, Float32Array, Float64Array
- Keyed collections - Map, Set, WeakMap, WeakSet
- Structured data - ArrayBuffer, DataView, JSON
- Control abstraction objects - Promise
- Reflection - Reflect, Proxy
- Internationalization - Intl
- Non-standard objects - Generator, Iterator, ParallelArray, StopIteration

The base should be provided to "parseInt"

Clave: javascript:S2427

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `parseInt` function has two versions, one that takes a base value as a second argument, and one that does not. Unfortunately using the single-arg version can result in unexpected results on older browsers.

Noncompliant code example

```
parseInt("010");  // Noncompliant; pre-2013 browsers may return 8
```

Compliant solution

```
parseInt("010", 10);
```

Object literal syntax should be used

Clave: javascript:S2428

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Object literal syntax, which initializes an object's properties inside the object declaration is cleaner and clearer than the alternative: creating an empty object, and then giving it properties one by one.

An issue is raised when the following pattern is met:

- An empty object is created.
- A consecutive single-line statement adds a property to the created object.

Noncompliant code example

```
let person = {};  // Noncompliant
person.firstName = "John";
person.middleInitial = "Q";
person.lastName = "Public";
```

Compliant solution

```
let person = {
  firstName: "John",
  middleInitial: "Q",
  lastName: "Public",
}
```

Constructor names should start with an upper case letter

Clave: javascript:S2430

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

By convention, constructor function names should start with an upper case letter as a reminder that they should be called with the `new` keyword.

A function is considered to be a constructor when it sets all of its arguments as object properties, and returns no value.

Noncompliant code example

```
function person (firstName, middleInitial, lastName) {  // Noncompliant
  this.firstName = firstName;
  this.middleInitial = middleInitial;
  this.lastName = lastName;
}
```

Compliant solution

```
function Person (firstName, middleInitial, lastName) {
  this.firstName = firstName;
  this.middleInitial = middleInitial;
  this.lastName = lastName;
}
```

Setters should not return values

Clave: javascript:S2432

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, a setter is a special type of function that is used to set the value of a property on an object. Setters are defined using the `set` keyword followed by the name of the property that the setter is associated with.

To set the property, we simply assign a value to it as if it were a regular property. The setter function is automatically called with the value that we assign to the property.

Functions declared with the `set` keyword will automatically return the values they were passed. Thus any value explicitly returned from a setter will be ignored, and explicitly returning a value is a mistake.

```
let person = {
  // ...
  set firstname(first) {
    this.first = first;
    return 42;  // Noncompliant: The return value 42 will be ignored
  }
};
console.log(person.firstname = 'bob'); // Prints 'bob'
```

Since return values in setters are ignored, you should remove return statements altogether.

```
let person = {
  // ...
  set firstname(first) {
    this.first = first;
  }
};
console.log(person.firstname = 'bob'); // Prints 'bob'
```

resources

Documentation

- MDN web docs - set

Setting loose POSIX file permissions is security-sensitive

Clave: javascript:S2612

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

The most restrictive possible permissions should be assigned to files and directories.

Compliant Solution

Node.js `fs`

```
const fs = require('fs');

fs.chmodSync("/tmp/fs", 0o770); // Compliant
```

```
const fs = require('fs');
const fsPromises = fs.promises;

fsPromises.chmod("/tmp/fsPromises", 0o770); // Compliant
```

```
const fs = require('fs');
const fsPromises = fs.promises

async function fileHandler() {
  let filehandle;
  try {
    filehandle = fsPromises.open('/tmp/fsPromises', 'r');
    filehandle.chmod(0o770); // Compliant
  } finally {
    if (filehandle !== undefined)
      filehandle.close();
  }
}
```

Node.js `process.umask`

```
const process = require('process');

process.umask(0o007); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP File Permission
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-266 - Incorrect Privilege Assignment

- STIG Viewer - Application Security and Development: V-222430 - The application must execute without excessive account permissions.

assess_the_problem

Ask Yourself Whether

- The application is designed to be run on a multi-user environment.
- Corresponding files and directories may contain confidential information.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Node.js `fs`

```
const fs = require('fs');

fs.chmodSync("/tmp/fs", 0o777); // Sensitive
```

```
const fs = require('fs');
const fsPromises = fs.promises;

fsPromises.chmod("/tmp/fsPromises", 0o777); // Sensitive
```

```
const fs = require('fs');
const fsPromises = fs.promises

async function fileHandler() {
  let filehandle;
  try {
    filehandle = fsPromises.open('/tmp/fsPromises', 'r');
    filehandle.chmod(0o777); // Sensitive
  } finally {
    if (filehandle !== undefined)
      filehandle.close();
  }
}
```

Node.js `process.umask`

```
const process = require('process');

process.umask(0o000); // Sensitive
```

default

In Unix file system permissions, the "`others`" category refers to all users except the owner of the file system resource and the members of the group assigned to this resource.

Granting permissions to this category can lead to unintended access to files or directories that could allow attackers to obtain sensitive information, disrupt services or elevate privileges.

Ask Yourself Whether

- The application is designed to be run on a multi-user environment.
- Corresponding files and directories may contain confidential information.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

The most restrictive possible permissions should be assigned to files and directories.

Sensitive Code Example

Node.js `fs`

```
const fs = require('fs');

fs.chmodSync("/tmp/fs", 0o777); // Sensitive
```

```
const fs = require('fs');
const fsPromises = fs.promises;

fsPromises.chmod("/tmp/fsPromises", 0o777); // Sensitive
```

```
const fs = require('fs');
const fsPromises = fs.promises

async function fileHandler() {
  let filehandle;
  try {
    filehandle = fsPromises.open('/tmp/fsPromises', 'r');
    filehandle.chmod(0o777); // Sensitive
  } finally {
    if (filehandle !== undefined)
      filehandle.close();
  }
}
```

Node.js `process.umask`

```
const process = require('process');

process.umask(0o000); // Sensitive
```

Compliant Solution

Node.js `fs`

```
const fs = require('fs');

fs.chmodSync("/tmp/fs", 0o770); // Compliant
```

```
const fs = require('fs');
const fsPromises = fs.promises;

fsPromises.chmod("/tmp/fsPromises", 0o770); // Compliant
```

```
const fs = require('fs');
const fsPromises = fs.promises

async function fileHandler() {
  let filehandle;
  try {
    filehandle = fsPromises.open('/tmp/fsPromises', 'r');
    filehandle.chmod(0o770); // Compliant
  } finally {
    if (filehandle !== undefined)
      filehandle.close();
  }
}
```

Node.js `process.umask`

```
const process = require('process');

process.umask(0o007); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP File Permission
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-266 - Incorrect Privilege Assignment
- STIG Viewer - Application Security and Development: V-222430 - The application must execute without excessive account permissions.

root_cause

In Unix file system permissions, the "`others`" category refers to all users except the owner of the file system resource and the members of the group assigned to this resource.

Granting permissions to this category can lead to unintended access to files or directories that could allow attackers to obtain sensitive information, disrupt services or elevate privileges.

Empty character classes should not be used

Clave: javascript:S2639

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Character classes
- MDN web docs - Regular expressions

root_cause

Character classes in regular expressions will match any of the characters enclosed in the square brackets (`[abc]` will match `a`, `b` or `c`).

You can specify a range of characters using a hyphen (`-`). If the hyphen appears as the first or last character, it will be matched as a literal hyphen.

An empty character class (`[]`) will not match any character because the set of matching characters is empty. So the regular expression will not work as you intended.

```
/^foo[]/.test(str); // Noncompliant: always returns "false"
```

Use a non-empty character class or a different regular expression pattern that achieves the desired result.

```
/^foo/.test(str);
```

"catch" clauses should do more than rethrow

Clave: javascript:S2737

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `try...catch`
- MDN web docs - `throw`

root_cause

A `catch` clause that only rethrows the caught exception has the same effect as omitting the `catch` altogether and letting it bubble up automatically.

```
try {
  doSomething();
} catch (ex) {  // Noncompliant
  throw ex;
}
```

Such clauses should either be removed or populated with the appropriate logic.

```
doSomething();
```

or

```
try {
  doSomething();
} catch (ex) {
  console.err(ex);
  throw ex;
}
```

XML parsers should not be vulnerable to XXE attacks

Clave: javascript:S2755

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

resources

Standards

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A4 - XML External Entities (XXE)
- CWE - CWE-611 - Information Exposure Through XML External Entity Reference
- CWE - CWE-827 - Improper Control of Document Type Definition
- STIG Viewer - Application Security and Development: V-222608 - The application must not be vulnerable to XML-oriented attacks.

introduction

This vulnerability allows the usage of external entities in XML.

root_cause

External Entity Processing allows for XML parsing with the involvement of external entities. However, when this functionality is enabled without proper precautions, it can lead to a vulnerability known as XML External Entity (XXE) attack.

What is the potential impact?

Exposing sensitive data

One significant danger of XXE vulnerabilities is the potential for sensitive data exposure. By crafting malicious XML payloads, attackers can reference external entities that contain sensitive information, such as system files, database credentials, or configuration files. When these entities are processed during XML parsing, the attacker can extract the contents and gain unauthorized access to sensitive data. This poses a severe threat to the confidentiality of critical information.

Exhausting system resources

Another consequence of XXE vulnerabilities is the potential for denial-of-service attacks. By exploiting the ability to include external entities, attackers can construct XML payloads that cause resource exhaustion. This can overwhelm the system’s memory, CPU, or other critical resources, leading to system unresponsiveness or crashes. A successful DoS attack can disrupt the availability of services and negatively impact the user experience.

Forging requests

XXE vulnerabilities can also enable Server-Side Request Forgery (SSRF) attacks. By leveraging the ability to include external entities, an attacker can make the vulnerable application send arbitrary requests to other internal or external systems. This can result in unintended actions, such as retrieving data from internal resources, scanning internal networks, or attacking other systems. SSRF attacks can lead to severe consequences, including unauthorized data access, system compromise, or even further exploitation within the network infrastructure.

how_to_fix

The following code contains examples of XML parsers that have external entity processing enabled. As a result, the parsers are vulnerable to XXE attacks if an attacker can control the XML file that is processed.

Noncompliant code example

```
var libxmljs = require('libxmljs');
var fs = require('fs');

var xml = fs.readFileSync('xxe.xml', 'utf8');
libxmljs.parseXmlString(xml, {
    noblanks: true,
    noent: true, // Noncompliant
    nocdata: true
});
```

Compliant solution

`parseXmlString` is safe by default.

```
var libxmljs = require('libxmljs');
var fs = require('fs');

var xml = fs.readFileSync('xxe.xml', 'utf8');
libxmljs.parseXmlString(xml);
```

How does this work?

Disable external entities

The most effective approach to prevent XXE vulnerabilities is to disable external entity processing entirely, unless it is explicitly required for specific use cases. By default, XML parsers should be configured to reject the processing of external entities. This can be achieved by setting the appropriate properties or options in your XML parser library or framework.

If external entity processing is necessary for certain scenarios, adopt a whitelisting approach to restrict the entities that can be resolved during XML parsing. Create a list of trusted external entities and disallow all others. This approach ensures that only known and safe entities are processed.
You should rely on features provided by your XML parser to restrict the external entities.

Non-existent operators '=+', '=-' and '=!' should not be used

Clave: javascript:S2757

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using operator pairs (`=+`, `=-`, or `=!`) that look like reversed single operators (`+=`, `-=` or `!=`) is confusing. They compile and run but do not produce the same result as their mirrored counterpart.

```
let target =-5;
let num = 3;

target =- num;  // Noncompliant: target = -3. Is that the expected behavior?
target =+ num; // Noncompliant: target = 3
```

This rule raises an issue when `=+`, `=-`, or `=!` are used without any space between the operators and when there is at least one whitespace after.

Replace the operators with a single one if that is the intention

```
let target =-5;
let num = 3;

target -= num;  // target = -8
```

Or fix the spacing to avoid confusion

```
let target =-5;
let num = 3;

target = - num;  // target = -3
```

"delete" should not be used on arrays

Clave: javascript:S2870

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - delete operator
- MDN web docs - Sparse arrays
- MDN web docs - Deleting array elements
- MDN web docs - Copying methods and mutating methods
- MDN web docs - Array.prototype.pop()
- MDN web docs - Array.prototype.shift()
- MDN web docs - Array.prototype.splice()

root_cause

The `delete` operator can be used to remove a property from any object. Arrays are objects, so the `delete` operator can be used on them too.

When you delete an element from an array using the `delete` keyword, it will remove the value but still leave behind an empty slot at that index. Therefore, a hole will be created in the array because the indexes won't be shifted to reflect the deletion. This means that the array will still have that index, but the value will be `undefined`.

Arrays that have gaps or missing indexes between elements are known as sparse arrays.

```
let myArray = ['a', 'b', 'c', 'd'];

delete myArray[2]; // Noncompliant: myArray => ['a', 'b', undefined, 'd']
console.log(myArray[2]); // expected value was 'd' but output is undefined
```

The proper method for removing an element from an array should be one of the following:

- `Array.prototype.splice()` - removes element(s) from an array at certain indexe(s)
- `Array.prototype.pop()` - removes the last element from an array
- `Array.prototype.shift()` - removes the first element from an array

Note that these methods mutate arrays in-place. Alternatively, you could create new arrays using copying methods and exclude the element you want to remove.

```
let myArray = ['a', 'b', 'c', 'd'];

// removes 1 element from index 2
removed = myArray.splice(2, 1);  // myArray => ['a', 'b', 'd']
console.log(myArray[2]); // outputs 'd'
```

"Array.prototype.sort()" and "Array.prototype.toSorted()" should use a compare function

Clave: javascript:S2871

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

JavaScript provides built-in methods to sort arrays, making it convenient for developers to manipulate data. There are two primary ways to sort an array:

- `Array.prototype.sort()`: This method sorts the elements of an array in place and returns the sorted array.
- `Array.prototype.toSorted()`: This method is designed to return a new sorted array, leaving the original array unchanged.

The default sort order is lexicographic (dictionary) order, based on the string representation of the elements. This means that when sorting an array of strings, numbers, or other elements, they are converted to strings and sorted according to their Unicode code points (UTF-16). For most cases, this default behavior is suitable when sorting an array of strings.

However, it's essential to be aware of potential pitfalls when sorting arrays of non-string elements, particularly numbers. The lexicographic order may not always produce the expected results for numbers:

```
const numbers = [10, 2, 30, 1, 5];
numbers.sort(); // Noncompliant: lexicographic sort
console.log(numbers); // Output: [1, 10, 2, 30, 5]
```

To sort numbers correctly, you must provide a custom comparison function that returns the correct ordering:

```
const numbers = [10, 2, 30, 1, 5];
numbers.sort((a, b) => a - b);
console.log(numbers); // Output: [1, 2, 5, 10, 30]
```

Even to sort strings, the default sort order may give unexpected results. Not only does it not support localization, it also doesn't fully support Unicode, as it only considers UTF-16 code units. For example, in the code below, `"eΔ"` is surprisingly before and after `"éΔ"`. To guarantee that the sorting is reliable and remains as such in the long run, it is necessary to provide a compare function that is both locale and Unicode aware - typically `String.localeCompare`.

```
const code1 = '\u00e9\u0394'; // "éΔ"
const code2 = '\u0065\u0301\u0394'; // "éΔ" using Unicode combining marks
const code3 = '\u0065\u0394'; // "eΔ"
console.log([code1, code2, code3].sort()); // Noncompliant: ["éΔ", "eΔ", "éΔ"], "eΔ" position is inconsistent
console.log([code1, code2, code3].sort((a, b) => a.localeCompare(b))); // ["eΔ", "éΔ", "éΔ"]
```

resources

Documentation

- MDN web docs - `Array.prototype.sort()`
- MDN web docs - `Array.prototype.toSorted()`
- MDN web docs - `String.prototype.localeCompare()`

Assertions should be complete

Clave: javascript:S2970

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Chai.js Documentation - Assert
- Chai.js Documentation - `expect` and `should`

root_cause

Assertions are statements that check whether certain conditions are true. They are used to validate that the actual results of a code snippet match the expected outcomes. By using assertions, developers can ensure that their code behaves as intended and identify potential bugs or issues early in the development process.

An incomplete assertion refers to a situation where an assertion is written but lacks some necessary components or conditions, making it insufficient to fully validate the expected behavior of the code being tested. Writing incomplete assertions can lead to false positives or false negatives in your test suite, making it less reliable.

This rule checks for incomplete assertions with Chai.js in the following cases:

- When `assert.fail`, `expect.fail` or `should.fail` are present but not called.
- When an `expect(...)` or `should` assertion is not followed by an assertion method, such as `equal`.
- When an `expect` or `should` assertion ends with a chainable getters, such as `.that`, or a modifier, such as `.deep`.
- When an `expect` or `should` assertion function, such as `.throw`, is not called.

In such cases, what is intended to be an assertion doesn't actually assert anything.

```
const assert = require('chai').assert;
const expect = require('chai').expect;

describe("incomplete assertions", function() {
    const value = 42;

    it("uses chai 'assert'", function() {
        assert.fail; // Noncompliant: Missing the call to 'fail'
    });

    it("uses chai 'expect'", function() {
        expect(1 == 1); // Noncompliant: Should chain with 'to.equal'
        expect(value.toString).to.throw; // Noncompliant: Missing the type of the exception
    });
});
```

Make sure to write complete and precise assertions. Always include the necessary comparison methods (e.g., `.to.equal()`, `.to.be.true`, etc.) to make the expectations clear and leave no room for ambiguity.

```
const assert = require('chai').assert;
const expect = require('chai').expect;

describe("complete assertions", function() {
    const value = 42;

    it("uses chai 'assert'", function() {
        assert.fail();
    });

    it("uses chai 'expect'", function() {
        expect(1).to.equal(1);
        expect(value.toString).to.throw(TypeError);
    });
});
```

The global "this" object should not be used

Clave: javascript:S2990

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - this
- MDN web docs - No this substitution
- MDN web docs - globalThis
- MDN web docs - Global object
- MDN web docs - Strict mode
- MDN web docs - Reflect.apply()
- MDN web docs - Function.prototype.call()
- MDN web docs - Function.prototype.apply()
- MDN web docs - Function.prototype.bind()

root_cause

The value of `this` depends on which context it appears:

- Function: The value of `this` will depend on how a function was called. The value of `this` is not always the object that has the function as an *own* property, but the object that is used to call the function. The methods `Function.prototype.call()`, `Function.prototype.apply()`, or `Reflect.apply()` can be used to explicitly set the value of `this`. Is it also possible to create a new function with a specific value of this that doesn't change regardless of how the function is called with `Function.prototype.bind()`. In non-strict mode, `this` will always be an object and will default to `globalThis` if set to `undefined` or `null`.
- Arrow function: The value of `this` will be the same as the enclosing context. Arrow functions will not create a new `this` binding. When invoking arrow functions using `call()`, `bind()`, or `apply()`, the `thisArg` parameter is ignored.
- Class: Class methods behave like methods in other objects: the `this` value is the object that the method was accessed on. If the method is not transferred to another object, `this` is generally an instance of the class. However, for static methods, the value of `this` is the class instead of the instance.

- Global: outside of any functions or classes (also inside blocks or arrow functions defined in the global scope), the value of `this` depends on what execution context the script runs in.

When a function is called without an explicit object context, the `this` keyword refers to the global object. This also applies when it is used outside of an object or function. The global `this` object refers to the global context in which the JavaScript code is executed. This can cause problems when the code is executed in different contexts, such as in a browser or in a Node.js environment, where the global object is different. Such uses could confuse maintainers as the actual value depends on the execution context, and it can be unclear what object the `this` keyword is referring to.

In JavaScript’s "strict mode", using `this` in the global context will always be `undefined`.

```
this.foo = 1;   // Noncompliant: 'this' refers to global 'this'
console.log(this.foo); // Noncompliant: 'this' refers to global 'this'

function MyObj() {
  this.foo = 1; // Compliant
}

MyObj.func1 = function() {
  if (this.foo === 1) { // Compliant
    // ...
  }
}
```

Instead, simply drop the `this`, or replace it with `globalThis`. The `globalThis` global property gives access to the global object regardless of the current environment.

```
foo = 1;
console.log(foo);

function MyObj() {
  this.foo = 1;
}

MyObj.func1 = function() {
  if (this.foo === 1) {
    // ...
  }
}
```

"new" should only be used with functions and classes

Clave: javascript:S2999

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `new` operator
- MDN web docs - Constructor
- MDN web docs - `TypeError: "x" is not a constructor`

root_cause

JavaScript has the `new` keyword that is used in conjunction with constructor functions to create new instances of objects. When you use the `new` keyword with a function, it signifies that the function is intended to be used as a constructor function to create objects.

Any function can be used as a constructor function by convention. Constructor functions are used to create new objects with the same structure or properties. They are typically named with an initial capital letter to distinguish them from regular functions.

To create a new instance of an object using the constructor function, you use the `new` keyword before the function call.

The `new` keyword should only be used with objects that define a constructor function. Attempting to use it with an object or a variable that is not a constructor will raise a `TypeError`.

```
function MyClass() {
  this.foo = 'bar';
}

const someClass = 1;

const obj1 = new someClass;    // Noncompliant: someClass is a variable
const obj2 = new MyClass();    // Noncompliant if parameter considerJSDoc is true. Compliant when considerJSDoc is false
```

Always use the `new` keyword with constructor functions or classes.

```
/**
 * @constructor
 */
function MyClass() {
  this.foo = 'bar';
}

const someClass = function(){
  this.prop = 1;
}

const obj1 = new someClass;
const obj2 = new MyClass();
```

Strings and non-strings should not be added

Clave: javascript:S3402

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Use a + with two numbers and you'll get addition. But use it with a string and anything else, and you'll get concatenation. This could be confusing, especially if it's not obvious that one of the operands is a string. It is recommended to explicitly convert the non-string component to make it easier to understand to future maintainers.

This rule raises an issue when + or += is used with a string and a non-string.

Noncompliant code example

```
function foo() {
  let x = 5 + 8;  // okay
  let z = "8"
  return x + z;  // Noncompliant; yields string "138"
}
```

Compliant solution

```
function foo() {
  let x = 5 + 8;
  let z = "8"
  return x + Number(z);
}
```

Strict equality operators should not be used with dissimilar types

Clave: javascript:S3403

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The strict equality operator in JavaScript is represented by three equal signs (===), the strict inequality with (!==). It is used to compare two values for equality, but with an important difference from the regular equality operator (==). The strict equality operator compares both value and type, while the regular equality operator only compares values after performing type coercion if necessary.

The problem with using the strict equality operator (===) with operands of dissimilar types lies in the way JavaScript handles the comparison. When you use === to compare two values of different types, it will always return false since their types are different, regardless of whether the values could be considered equal under certain conditions.

Noncompliant code example

```
let a = 8;
let b = "8";

if (a === b) { // Noncompliant: Always false since 'a' is a number and 'b' a string
  // ...
}
```

Compliant solution

To address this issue, you can use the loose equality operator (==), which performs type coercion.

```
let a = 8;
let b = "8";

if (a == b) {
  // ...
}
```

Alternatively, use the strict equality operator (`===`) but ensure that the operands have the same type before performing the comparison. You can explicitly convert the operands to a common type using functions like `Number()`, `String()`, or other appropriate methods depending on the situation.

```
let a = 8;
let b = "8";

if (a === Number(b)) {
  // ...
}
```

resources

Documentation

- MDN - Strict equality (`===`)
- MDN - Strict inequality (`!==`)
- MDN - Equality (`==`)
- MDN - Inequality (`!=`)
- MDN - Type coercion
- MDN - `Number()` constructor
- MDN - `String()` constructor
- MDN - `Boolean()` constructor

introduction

This rule raises an issue when a strict equality operator is used to compare objects of different types.

"const" variables should not be reassigned

Clave: javascript:S3500

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `const`
- MDN web docs - `let`
- MDN web docs - `Object.freeze()`
- Wikipedia - ECMAScript 2015

root_cause

Variables declared with `const` cannot be reassigned using the assignment operator.

The `const` declaration creates an immutable reference to a value. This does not mean the value it holds is immutable, but the identifier cannot be reassigned. For example, if the constant is an object, its properties can still be altered. Use `Object.freeze()` to make an object immutable.

You must always specify an initializer in a `const` declaration as it can not be changed later. Trying to declare a constant without an initializer (`const foo;`) will throw a SyntaxError.

Trying to reassign a constant will throw a TypeError. In a non-ES2015 environment, it might simply be ignored.

```
const pi = 3.14;
pi = 3.14159; // Noncompliant: TypeError: invalid assignment to const 'pi'
```

If a variable will need to be reassigned, use `let` instead.

```
let pi = 3.14;
pi = 3.14159;
```

Variables should be declared with "let" or "const"

Clave: javascript:S3504

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Variables declared with `var` are function-scoped, meaning they are accessible within the entire function in which they are defined. If a variable is declared using `var` outside of any function, it becomes a global variable and is accessible throughout the entire JavaScript program.

`let` and `const` were introduced in ECMAScript 6 (ES6) as a block-scoped alternative to `var`. Variables declared with `let` have block scope, meaning they are limited to the block of code in which they are defined. A block is typically delimited by curly braces `{}`.

Variables declared with `const` are also block-scoped, similar to `let`. However, `const` variables are immutable, meaning their value cannot be changed after assignment. This applies to the binding between the variable name and its value, but it does not mean the value itself is immutable if it is an object or an array.

A variable declared with `let` or `const` is said to be in a "temporal dead zone", meaning the period between entering a scope and declaring a `let` or `const` variable. During this phase, accessing the variable results in a `ReferenceError`. This helps catch potential errors and encourages proper variable declaration.

Unlike `let` and `const`, variables declared with `var` are subject to "hoisting", which means that they are moved to the top of their scope during the compilation phase, even if the actual declaration is placed lower in the code.

Hoisting can sometimes lead to unexpected behavior. For example, variables declared with `var` are accessible before they are declared, although they will have the value `undefined` until the declaration is reached.

The distinction between the variable types created by `var` and by `let` is significant, and a switch to `let` will help alleviate many of the variable scope issues which have caused confusion in the past.

Because these new keywords create more precise variable types, they are preferred in environments that support ECMAScript 2015. However, some refactoring may be required by the switch from `var` to `let`, and you should be aware that they raise `SyntaxErrors` in pre-ECMAScript 2015 environments.

This rule raises an issue when `var` is used instead of `const` or `let`.

```
var color = "blue"; // Noncompliant
var size = 4;       // Noncompliant
```

You should declare your variables with either `const` or `let` depending on whether you are going to modify them afterwards.

```
const color = "blue";
let size = 4;
```

resources

Documentation

- MDN web docs - var
- MDN web docs - let
- MDN web docs - const
- MDN web docs - Scope
- MDN web docs - Hoisting
- MDN web docs - Temporal dead zone (TDZ)

Template strings should be used instead of concatenation

Clave: javascript:S3512

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ECMAScript 2015 added the ability to use template literals instead of concatenation. Since their use is clearer and more concise, they are preferred in environments that support ECMAScript 2015.

Noncompliant code example

```
function sayHello(name) {
  console.log("hello " + name);  // Noncompliant
}
```

Compliant solution

```
function sayHello(name) {
  console.log(`hello ${name}`);
}
```

"arguments" should not be accessed directly

Clave: javascript:S3513

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The magic of JavaScript is that you can pass arguments to functions that don’t declare parameters, and on the other side, you can use those passed-in arguments inside the no-args `function`.

But just because you can, that does’t mean you should. The expectation and use of arguments inside functions that don’t explicitly declare them is confusing to callers. No one should ever have to read and fully understand a function to be able to use it competently.

If you don’t want to name arguments explicitly, use the `...` syntax to specify that an a variable number of arguments is expected. Then inside the function, you’ll be dealing with a first-class array, rather than an array-like structure.

Noncompliant code example

```
function concatenate() {
  let args = Array.prototype.slice.call(arguments);  // Noncompliant
  return args.join(', ');
}

function doSomething(isTrue) {
  var args = Array.prototype.slice.call(arguments, 1); // Noncompliant
  if (!isTrue) {
    for (var arg of args) {
      ...
    }
  }
}
```

Compliant solution

```
function concatenate(...args) {
  return args.join(', ');
}

function doSomething(isTrue, ...values) {
  if (!isTrue) {
    for (var value of values) {
      ...
    }
  }
}
```

Destructuring syntax should be used for assignments

Clave: javascript:S3514

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ECMAScript 2015 introduced the ability to extract and assign multiple data points from an object or array simultaneously. This is called "destructuring", and it allows you to condense boilerplate code so you can concentrate on logic.

This rule raises an issue when multiple pieces of data are extracted out of the same object or array and assigned to variables.

Noncompliant code example

```
function foo (obj1, obj2, array) {
  var a = obj1.a;  // Noncompliant
  var b = obj1.b;

  var name = obj2.name;  // ignored; there's only one extraction-and-assignment
  var zero = array[0];  // Noncompliant
  var one = array[1];
}
```

Compliant solution

```
function foo (obj1, obj2, array) {
  var {a, b} = obj1;

  var {name} = obj2;  // this syntax works because var name and property name are the same

  var [zero, one] = array;
}
```

Function returns should not be invariant

Clave: javascript:S3516

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a function has multiple `return` statements and returns the same value in more than one of them, it can lead to several potential problems:

- It can make the code more difficult to understand and maintain, as the reader may be unsure why the same value is being returned multiple times. This can introduce ambiguity and increase the chances of misunderstanding the function’s intent.
- The use of multiple return statements with the same value might lead someone to assume that each return corresponds to a distinct case or outcome. However, if they all return the same value, it can be misleading and may indicate an oversight or mistake in the code.
- When the function needs to be modified or extended in the future, having multiple identical return statements can make it harder to implement changes correctly across all occurrences. This can introduce bugs and inconsistencies in the codebase.
- Code readability is crucial for maintainability and collaboration. Having repetitive return statements can lead to unnecessary code duplication, which should be avoided in favor of creating modular and clean code.

This rule raises an issue when a function contains several `return` statements that all return the same value.

```
function f(a, g) { // Noncompliant: 'f' returns 'b' on two different return statements
  const b = 42;
  if (a) {
    g(a);
    return b;
  }
  return b;
}
```

To address this, you should refactor the function to use a single return statement with a variable storing the value to be returned. This way, the code becomes more concise, easier to understand, and reduces the likelihood of introducing errors when making changes in the future. By using a single return point, you can also enforce consistency and prevent unexpected return values.

```
function f(a, g) {
  const b = 42;
  if (a) {
    g(a);
  }
  return b;
}
```

resources

Documentation

- MDN web docs - `return`
- MDN web docs - Function return values

Function constructors should not be used

Clave: javascript:S3523

Severidad: MAJOR

Estado: ⚠️ Depreciada

Descripción: No disponible

root_cause

In addition to being obtuse from a syntax perspective, function constructors are also dangerous: their execution evaluates the constructor’s string arguments similar to the way `eval` works, which could expose your program to random, unintended code which can be both slow and a security risk.

In general it is better to avoid it altogether, particularly when used to parse JSON data. You should use ECMAScript 5’s built-in JSON functions or a dedicated library.

Noncompliant code example

```
var obj =  new Function("return " + data)();  // Noncompliant
```

Compliant solution

```
var obj = JSON.parse(data);
```

Exceptions

Function calls where the argument is a string literal (e.g. `(Function('return this'))()`) are ignored.

resources

- OWASP - Top 10 2017 Category A1 - Injection

introduction

This rule is deprecated; use S1523 instead.

Braces and parentheses should be used consistently with arrow functions

Clave: javascript:S3524

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Arrow functions in JavaScript provide a concise syntax to write function expressions. However, the use of braces `{}` and parentheses `()` should be consistent in arrow functions for the following reasons:

- Readability: Consistent use of braces and parentheses improves the readability of the code. It makes it easier for other developers to understand the code quickly and reduces the chances of misinterpretation.
- Predictability: When braces and parentheses are used consistently, it makes the code more predictable. Developers can easily predict the outcome of the function.
- Avoid Errors: Inconsistent use of braces and parentheses can lead to errors. For example, if braces are omitted for a function that has more than one statement, it will result in a syntax error.
- Code Maintenance: Consistent use of braces and parentheses makes the code easier to maintain. It’s easier to add or remove code lines without worrying about adjusting braces or parentheses.

Shared coding conventions allow teams to collaborate effectively. This rule raises an issue when using parentheses and curly braces with an arrow function does not conform to the configured requirements.

resources

Documentation

- MDN web docs - Arrow function expressions

how_to_fix

Use parentheses and curly braces with arrow functions consistently. By default, the rule forbids arrow functions to have parentheses around single parameters and curly braces around single-return bodies.

Noncompliant code example

```
const foo = (a) => { /* ... */ };  // Noncompliant; remove the parentheses from the parameter
const bar = (a, b) => { return 0; };  // Noncompliant; remove the curly braces from the body
```

Compliant solution

```
const foo = a => { /* ... */ };
const bar = (a, b) => 0;
```

Class methods should be used instead of "prototype" assignments

Clave: javascript:S3525

Severidad: CRITICAL

Estado: ☑ Activa

Descripción: No disponible

root_cause

Originally JavaScript didn’t support `classes`, and class-like behavior had to be kludged using things like `prototype` assignments for "class" functions. Fortunately, ECMAScript 2015 added classes, so any lingering `prototype` uses should be converted to true `classes`. The new syntax is more expressive and clearer, especially to those with experience in other languages.

Specifically, with ES2015, you should simply declare a `class` and define its methods inside the class declaration.

Noncompliant code example

```
function MyNonClass(initializerArgs = []) {
  this._values = [...initializerArgs];
}

MyNonClass.prototype.doSomething = function () {  // Noncompliant
  // ...
}
```

Compliant solution

```
class MyClass {
  constructor(initializerArgs = []) {
    this._values = [...initializerArgs];
  }

  doSomething() {
    //...
  }
}
```

Comma and logical OR operators should not be used in switch cases

Clave: javascript:S3616

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

The use of the comma operator and logical OR (||) operator within switch cases is not recommended. The switch statement is designed to evaluate a single expression and compare it against multiple values. When you use the comma operator or logical OR operator within a case, you’re essentially trying to match multiple values or conditions simultaneously: only the rightmost value will ever be considered with the comma operator, and the first truthy operand will be handled with the logical OR operator.

This behavior is not well-defined and can lead to unexpected results.

```
switch (a) {
  case 1, 2:   // Noncompliant: only 2 is matched by this case
    doTheThing(a);
  case 3 || 4: // Noncompliant: only 3 is matched by this case
    doThatThing(a);
  case 5:
    doTheOtherThing(a);
  default:
    console.log('Neener, neener!');
}
```

Using the comma operator or logical OR operator to combine multiple values or conditions within a single case can make the code more complex and difficult to read. It goes against the intention of the switch statement, which is to provide a concise and clear structure for branching based on a single value.

The switch statement should solely be used to rely on exact value matching instead.

```
switch (a) {
  case 1:
  case 2:
    doTheThing(a);
  case 3:
```

```
  case 4:
    doThatThing(a);
  case 5:
    doTheOtherThing(a);
  default:
    console.log('Neener, neener!');
}
```

The rule makes an exception for the `switch (true)` pattern, which is sometimes used as a workaround to achieve a similar effect to a series of `if-else` statements. This pattern allows you to evaluate multiple conditions acting as guards and execute corresponding code blocks based on the first matching condition.

```
function weekStatus (day) {
  let status;
  switch (true) {
    case (day === 'MON' || day === 'TUE' || day === 'WED' || day === 'THU' || day === 'FRI'):
      status = 'Weekday';
      break;
    case (day === 'SAT' || day === 'SUN'):
      status = 'Weekend';
      break;
  }
  return status;
}
```

resources

Documentation

- MDN web docs - `switch`
- MDN web docs - Comma operator `(,)`
- MDN web docs - Logical OR `(||)`

Jump statements should not be redundant

Clave: javascript:S3626

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `continue`
- MDN web docs - `break`
- MDN web docs - `return`

root_cause

Jump statements, such as `return`, `break` and `continue`, are used to change the normal flow of execution in a program. They are useful because they allow for more complex and flexible code. However, it is important to use jump statements judiciously, as overuse or misuse can make code difficult to read and maintain.

Jump statements are redundant when they do not affect the program flow or behavior.

```
function redundantJump(x) {
  if (x == 1) {
    console.log("x == 1");
    return; // Noncompliant: The function would return 'undefined' also without this 'return' statement
  }
}
```

Remove any jump statements that are unnecessary or redundant.

```
function redundantJump(x) {
  if (x == 1) {
    console.log("x == 1");
  }
}
```

Exceptions

- `break` and `return` inside `switch` statements are ignored, because they are often used for consistency.
- `continue` associated with a label is ignored, because it is usually used for clarity.
- Jump statements are ignored when they are the only statement inside a block.

Trailing commas should be used

Clave: javascript:S3723

Severidad: MAJOR

Estado: ⚠ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

Modern browsers ignore unneeded, trailing commas, so there are no negatives to having them unless you're supporting an IE 8 application. Since they make adding new properties simpler, their use is preferred. This rule raises an issue when the last item in a multiline construct (array or object literal, import or export statement, function declaration or call) does not end with a trailing comma and does not lie on the same line as the closing curly brace, bracket or parenthesis.

Noncompliant code example

```
var joe = {
  fname: "Joe",
  lname: "Smith"      // Noncompliant
};
```

Compliant solution

```
var joe = {
  fname: "Joe",
  lname: "Smith",    // OK
};

var joe = {
  fname: "Joe",
  lname: "Smith"};   // OK
```

"void" should not be used

Clave: javascript:S3735

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `void` operator
- MDN web docs - `undefined`
- MDN web docs - IIFE (Immediately Invoked Function Expression)

root_cause

The `void` operator evaluates its argument and always returns `undefined`. `void` allows using any expression where an `undefined` is expected. However, using `void` makes code more difficult to understand, as the intent is often unclear.

```
if (parameter === void 42) { // Noncompliant
   // ...
}
doSomethingElse(void doSomething()); // Noncompliant
```

Instead of using `void` to get the `undefined` value, use the `undefined` global property. In ECMAScript5 and newer environments, `undefined` cannot be reassigned. In other cases, remove the `void` operator to avoid confusion for maintainers.

```
if (parameter === undefined) {
   // ...
}
doSomething();
doSomethingElse();
```

Exceptions

- `void 0` (or the equivalent `void(0)`) is allowed as it was a conventional way to obtain the `undefined` value in environments before ECMAScript 5.

```
if (parameter === void 0) {
   // ...
}
```

- `void` is allowed with immediately invoked function expressions.

```
void function() {
   // ...
}();
```

- `void` is allowed with Promise-like objects to mark a promise as intentionally not awaited, as advised by @typescript-eslint/no-floating-promises.

```
const runPromise = () => Promise.resolve();
void runPromise();
```

Arithmetic operations should not result in "NaN"

Clave: javascript:S3757

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The result of an expression with an arithmetic operator `/, *, %, ++, --, -, +=, -=, *=, /=, %=, +` or unary operator `+, -` when at least one operand is `Object` or `Undefined` will be always a `NaN` (Not a Number).

Noncompliant code example

```
x = [1, 2];
var y = x / 4;  //Noncompliant
```

Exceptions

- `Date` operands: they are implicitly converted to numbers.
- The binary `+` operator with `Object` operand (concatenation).

Values not convertible to numbers should not be used in numeric comparisons

Clave: javascript:S3758

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In a Zen-like manner, `NaN` isn’t equal to anything, even itself. So comparisons (`>, <, >=, <=`) where one operand is `NaN` or evaluates to `NaN` always return `false`. Specifically, `undefined` and objects that cannot be converted to numbers evaluate to `NaN` when used in numerical comparisons.

This rule raises an issue when there is at least one path through the code where one of the operands to a comparison is `NaN`, `undefined` or an `Object` which cannot be converted to a number.

Noncompliant code example

```
var x;  // x is currently "undefined"
if (someCondition()) {
  x = 42;
}

if (42 > x) {  // Noncompliant; "x" might still be "undefined"
  doSomething();
}

var obj = {prop: 42};
if (obj > 24) { // Noncompliant
  doSomething();
}
```

Compliant solution

```
var x;
if (someCondition()) {
  x = 42;
} else {
  x = foo();
}

if (42 > x) {
  doSomething();
}

var obj = {prop: 42};
if (obj.prop > 24) {
  doSomething();
}
```

Arithmetic operators should only have numbers as operands

Clave: javascript:S3760

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Expressions with arithmetic (`/, *, %, ++, --, -, -=, *=, /=, %=, +=, +`), unary (`-`), or comparison operators (`>, <, >=, <=`) where one, or both, of the operands is a String, Boolean or Date value rely on implicit conversions. Both the maintainability and reliability levels of such a piece of code are questionable.

Noncompliant code example

```
str = "80";
quarter = str / 4; // Noncompliant

if (str < 10) { // Noncompliant
   // ...
}
```

Compliant solution

```
str = "80";
parsedStr = parseInt(str);
quarter = parsedStr / 4;

if (parsedStr < 10) {
  // ...
}
```

Exceptions

- Expressions using the binary `+` operator with at least one `String` operand are ignored because the `+` operator will perform a concatenation in that case.
- Comparisons where both operands are strings are ignored because a lexicographical comparison is performed in that case.

Parentheses should be used when negating "in" and "instanceof" operations

Clave: javascript:S3812

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Operator precedence
- MDN web docs - Operator precedence table
- MDN web docs - Logical NOT (`!`)
- MDN web docs - `instanceof`
- MDN web docs - `in` operator

root_cause

Operator precedence determines the order in which different operators are evaluated when an expression contains multiple ones. It helps determine how the expression is parsed and executed. JavaScript follows a specific set of rules to determine operator precedence.

Not being aware of JavaScript's operator precedence rules can lead to unexpected and potentially incorrect results when evaluating expressions. This is common when misapplying the logical negation operator (`!`). For instance, consider the difference between `!key in dict` and `!(key in dict)`. The first looks for a boolean value (`!key`) in `dict`, and the other looks for a string and inverts the result. The same applies for `!obj instanceof SomeClass`.

This rule raises an issue when the left operand of an `in` or `instanceof` operator is negated with `!`.

```
if (!"prop" in myObj) { // Noncompliant: checks whether !"prop", that is, false is in myObj
  doTheThing(); // this block is never executed
}

if (!foo instanceof MyClass) { // Noncompliant: "!foo" returns a boolean, which is not an instance of anything
  doTheOtherThing(); // this block is never executed either
}
```

You should use parentheses to force the order of evaluation of expressions mixing negation and `in` or `instanceof` operators.

```
if (!("prop" in myObj)) {
  doTheThing();
}

if (!(foo instanceof MyClass)) {
  doTheOtherThing();
}
```

Variables should be defined before being used

Clave: javascript:S3827

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a non-existent variable is referenced a `ReferenceError` is raised.

Due to the dynamic nature of JavaScript this can happen in a number of scenarios:

- When typo was made in a symbol's name.
- When using variable declared with `let` or `const` before declaration (unlike `var`-declarations, they are not hoisted to the top of the scope).
- Due to confusion with scopes of `let`- and `const`-declarations (they have block scope, unlike `var`-declarations, having function scope).
- When accessing a property in the wrong scope (e.g. forgetting to specify `this.`).

This rule does not raise issues on global variables which are defined with `sonar.javascript.globals` and `sonar.javascript.environments` properties.

Noncompliant code example

```
var john = {
  firstName: "john",
  show: function() { console.log(firstName); } // Noncompliant: firstName is not defined
}
john.show();
```

Compliant solution

```
var john = {
  firstName: "john",
  show: function() { console.log(this.firstName); }
}
john.show();
```

"new" operator should not be used with Symbol and BigInt

Clave: javascript:S3834

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using the `new` operator with `Symbol` and `BigInt` will throw a `TypeError` because they are not intended to be used as constructors. `Symbol` and `BigInt` are primitive types in JavaScript and should be used as such.

This is different from the other primitive types, such as string, number or boolean, where it was possible to call global `String` or `Number` as functions that return primitive types, but also use them as constructors with the `new` operator to create wrapper objects. This confusing double behavior is not implemented for `Symbol` and `BigInt` types that were introduced later in the language.

This behavior would be especially problematic for symbols that have reference identity and already behave like objects in some way. For example, they are garbage collectable and therefore can be used as keys in WeakMap and WeakSet objects.

```
let foo = new Symbol('abc'); // Noncompliant: TypeError: Symbol is not a constructor
let bar = new BigInt(123);   // Noncompliant: TypeError: BigInt is not a constructor
```

To fix the code remove the `new` operator.

```
let foo = Symbol('abc');
let bar = BigInt(123);
```

For the `BigInt` type to be recognized correctly, the environment should be `es2020` or higher.

resources

Documentation

- MDN web docs - Symbol
- MDN web docs - BigInt
- MDN web docs - new operator

"super()" should be invoked appropriately

Clave: javascript:S3854

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

In JavaScript, the `super` keyword is used to call the constructor and methods of an object’s parent class, and to access its properties.

The expression `super(...args)` is used to call the parent’s constructor. It must be used carefully and correctly to avoid errors.

```
class Dog extends Animal {
  constructor(name) {
    super();
    this.name = name;
    super(); // Noncompliant: constructor is called twice.
    super.doSomething();
  }
}
```

Follow these instructions when invoking the parent’s constructor:

- `super()` cannot be invoked in the constructor of a non-derived class.
- `super()` must be invoked in the constructor of a derived class.
- `super()` must be invoked before the `this` and `super` keywords can be used.
- `super()` must be invoked with the same number of arguments as the base class' constructor.
- `super()` can only be invoked in a constructor - not in any other method.
- `super()` cannot be invoked multiple times in the same constructor.

```
class Dog extends Animal {
  constructor(name) {
    super();
    this.name = name;
    super.doSomething();
  }
}
```

Some issues are not raised if the base class is not defined in the same file as the current class. This is a known limitation of this rule.

resources

Documentation

- MDN web docs - `super`
- MDN web docs - Classes
- MDN web docs - Inheritance and the prototype chain

Imports from the same module should be merged

Clave: javascript:S3863

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `import`
- MDN web docs - JavaScript modules

root_cause

Having the same module imported multiple times can affect code readability and maintainability. It makes hard to identify which modules are being used.

```
import { B1 } from 'b';
import { B2 } from 'b'; // Noncompliant: there is already an import from module 'b'.
```

Instead, one should consolidate the imports from the same module into a single statement. By consolidating all imports from the same module in a single `import` statement, the code becomes more concise and easier to read, as there is only one import statement to keep track of. Additionally, it can make it easier to identify which modules are used in the code.

```
import { B1, B2 } from 'b';
```

Conditionals should start on new lines

Clave: javascript:S3972

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Placing an `if` statement on the same line as the closing `}` from a preceding `if`, `else`, or `else if` block can lead to confusion and potential errors. It may indicate a missing `else` statement or create ambiguity for maintainers who might fail to understand that the two statements are unconnected.

The following code snippet is confusing:

```
if (condition1) {
  // ...
} if (condition2) {  // Noncompliant
  //...
}
```

Either the two conditions are unrelated and they should be visually separated:

```
if (condition1) {
  // ...
}

if (condition2) {
  //...
}
```

Or they were supposed to be exclusive and you should use `else if` instead:

```
if (condition1) {
  // ...
} else if (condition2) {
  //...
}
```

A conditionally executed single line should be denoted by indentation

Clave: javascript:S3973

Severidad: CRITICAL

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

When the line immediately after a conditional has neither curly braces nor indentation, the intent of the code is unclear and perhaps not what is executed. Additionally, such code is confusing to maintainers.

```
if (condition)  // Noncompliant
doTheThing();
doTheOtherThing(); // Was the intent to call this function unconditionally?
```

It becomes even more confusing and bug-prone if lines get commented out.

```
if (condition)  // Noncompliant
//  doTheThing();
doTheOtherThing(); // Was the intent to call this function conditionally?
```

Indentation alone or together with curly braces makes the intent clear.

```
if (condition)
  doTheThing();
doTheOtherThing(); // Clear intent to call this function unconditionally

// or

if (condition) {
  doTheThing();
}
doTheOtherThing(); // Clear intent to call this function unconditionally
```

This rule raises an issue if the line controlled by a conditional has the same indentation as the conditional and is not enclosed in curly braces.

introduction

This rule is deprecated, and will eventually be removed.

Collection size and array length comparisons should make sense

Clave: javascript:S3981

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The size of a collection and the length of an array are always greater than or equal to zero. Testing it doesn't make sense, since the result is always `true`.

```
if (someSet.size >= 0) {...} // Noncompliant always true
```

```
const result = someArray.length >= 0;  // Noncompliant always true
```

Similarly testing that it is less than zero will always return `false`.

```
if (someMap.size < 0) {...} // Noncompliant always false
```

Fix the code to properly check for emptiness if it was the intent, or remove the redundant code to keep the current behavior.

Errors should not be created without being thrown

Clave: javascript:S3984

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Error
- MDN web docs - throw

- MDN web docs - Control flow and error handling

root_cause

Errors should not be created without being thrown because they can confuse and make it difficult to debug code. When an error is thrown, it means that something unexpected has happened, and the program cannot continue executing as expected. By creating an error without throwing it, it may appear as if everything is working correctly, but in reality, an underlying issue must be addressed.

```
if (x < 0) {
  new Error("x must be nonnegative"); // Noncompliant: Creating an error without throwing it
}
```

You should make sure to always throw an error that you create using the `throw` keyword.

```
if (x < 0) {
  throw new Error("x must be nonnegative");
}
```

Media elements should have captions

Clave: javascript:S4084

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Captions in HTML media elements are text versions of the audio content, synchronized with the video. They are essential for individuals who are deaf or hard of hearing, as they provide a text alternative for the audio information. They can also be beneficial for individuals who are not native speakers of the language of the video, or for situations where the audio cannot be heard.

In the context of accessibility, providing captions for media elements is a requirement under the Web Content Accessibility Guidelines (WCAG). Without captions, you are excluding a portion of your audience who rely on them to understand the content of your media.

how_to_fix

If captions are missing from your media elements, you can fix this by adding a `<track>` element with the `kind="captions"` attribute inside your `<audio>` or `<video>` element. However, for video elements that have the `muted` attribute, captions are not required.

Noncompliant code example

```
<audio {...props} />; // Noncompliant
<video {...props} />; // Noncompliant
```

Compliant solution

```
<audio><track kind="captions" {...props} /></audio>
<video><track kind="captions" {...props} /></video>
<video muted {...props} ></video>
```

resources

Documentation

- MDN web docs - audio element
- MDN web docs - video element
- MDN web docs - track element
- WCAG - Captions
- WCAG - Audio Description or Media Alternative

"in" should not be used on arrays

Clave: javascript:S4619

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `in` operator is used to check if a property is in an object or its prototype chain.

When used on an array, it will compare against the indexes of the array, not the values. This is likely not to be the expected behavior.

```
function func() {
    const arr = ["a", "b", "c"];

    const expectedValue = "b";
    if (expectedValue in arr) { // Noncompliant: will be always false
        return expectedValue + " found in the array";
    } else {
        return expectedValue + " not found";
    }
}
```

Use the method `Array.prototype.includes()` to determine whether an array contains a certain value. If the actual intention is to check for an array slot, use `Object.prototype.hasOwnProperty()`.

```
function func() {
    const arr = ["a", "b", "c"];

    const expectedValue = "b";
    if (arr.includes(expectedValue)) {
        return expectedValue + " found in the array";
    } else {
        return expectedValue + " not found";
    }
}
```

resources

Documentation

- MDN web docs - `in` operator
- MDN web docs - `for...in`
- MDN web docs - `Array.prototype.includes()`

Using shell interpreter when executing OS commands is security-sensitive

Clave: javascript:S4721

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Arbitrary OS command injection vulnerabilities are more likely when a shell is spawned rather than a new process, indeed shell meta-chars can be used (when parameters are user-controlled for instance) to inject OS commands.

Ask Yourself Whether

- OS command name or parameters are user-controlled.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Use functions that don’t spawn a shell.

Sensitive Code Example

```
const cp = require('child_process');

// A shell will be spawn in these following cases:
cp.exec(cmd); // Sensitive
cp.execSync(cmd); // Sensitive

cp.spawn(cmd, { shell: true }); // Sensitive
cp.spawnSync(cmd, { shell: true }); // Sensitive
cp.execFile(cmd, { shell: true }); // Sensitive
cp.execFileSync(cmd, { shell: true }); // Sensitive
```

Compliant Solution

```
const cp = require('child_process');
```

```
cp.spawnSync("/usr/bin/file.exe", { shell: false }); // Compliant
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-78 - Improper Neutralization of Special Elements used in an OS Command

how_to_fix

Recommended Secure Coding Practices

Use functions that don’t spawn a shell.

Compliant Solution

```
const cp = require('child_process');

cp.spawnSync("/usr/bin/file.exe", { shell: false }); // Compliant
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-78 - Improper Neutralization of Special Elements used in an OS Command

assess_the_problem

Ask Yourself Whether

- OS command name or parameters are user-controlled.

There is a risk if you answered yes to this question.

Sensitive Code Example

```
const cp = require('child_process');

// A shell will be spawn in these following cases:
cp.exec(cmd); // Sensitive
cp.execSync(cmd); // Sensitive

cp.spawn(cmd, { shell: true }); // Sensitive
cp.spawnSync(cmd, { shell: true }); // Sensitive
cp.execFile(cmd, { shell: true }); // Sensitive
cp.execFileSync(cmd, { shell: true }); // Sensitive
```

root_cause

Arbitrary OS command injection vulnerabilities are more likely when a shell is spawned rather than a new process, indeed shell meta-chars can be used (when parameters are user-controlled for instance) to inject OS commands.

Using regular expressions is security-sensitive

Clave: javascript:S4784

Severidad: CRITICAL

Estado: ⚠ Depreciada

Descripción: No disponible

default

This rule is deprecated; use S5852 instead.

Using regular expressions is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-16021
- CVE-2018-13863

Evaluating regular expressions against input strings is potentially an extremely CPU-intensive task. Specially crafted regular expressions such as `(a+)+s` will take several seconds to evaluate the input string `aaaaaaaaaaaaaaaaaaaaaaaaaaaaabs`. The problem is that with every additional `a` character added to the input, the time required to evaluate the regex doubles. However, the equivalent regular expression, `a+s` (without grouping) is efficiently evaluated in milliseconds and scales linearly with the input size.

Evaluating such regular expressions opens the door to Regular expression Denial of Service (ReDoS) attacks. In the context of a web application, attackers can force the web server to spend all of its resources evaluating regular expressions thereby making the service inaccessible to genuine users.

This rule flags any execution of a hardcoded regular expression which has at least 3 characters and at least two instances of any of the following characters: `*+{`.

Example: `(a+)*`

Ask Yourself Whether

- the executed regular expression is sensitive and a user can provide a string which will be analyzed by this regular expression.
- your regular expression engine performance decrease with specially crafted inputs and regular expressions.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Check whether your regular expression engine (the algorithm executing your regular expression) has any known vulnerabilities. Search for vulnerability reports mentioning the one engine you’re are using.

Use if possible a library which is not vulnerable to Redos Attacks such as Google Re2.

Remember also that a ReDos attack is possible if a user-provided regular expression is executed. This rule won’t detect this kind of injection.

Sensitive Code Example

```
const regex = /(a+)+b/; // Sensitive
const regex2 = new RegExp("(a+)+b"); // Sensitive

str.search("(a+)+b"); // Sensitive
str.match("(a+)+b"); // Sensitive
str.split("(a+)+b"); // Sensitive
```

Note: String.matchAll does not raise any issue as it is not supported by NodeJS.

Exceptions

Some corner-case regular expressions will not raise an issue even though they might be vulnerable. For example: `(a|aa)+`, `(a|a?)+`.

It is a good idea to test your regular expression if it has the same pattern on both side of a "`|`".

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-624 - Executable Regular Expression Error
- OWASP Regular expression Denial of Service - ReDoS

root_cause

This rule is deprecated; use S5852 instead.

Using regular expressions is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-16021
- CVE-2018-13863

Evaluating regular expressions against input strings is potentially an extremely CPU-intensive task. Specially crafted regular expressions such as `(a+)+s` will take several seconds to evaluate the input string `aaaaaaaaaaaaaaaaaaaaaaaaaaaaabs`. The problem is that with every additional `a` character added to the input, the time required to evaluate the regex doubles. However, the equivalent regular expression, `a+s` (without grouping) is efficiently evaluated in milliseconds and scales linearly with the input size.

Evaluating such regular expressions opens the door to Regular expression Denial of Service (ReDoS) attacks. In the context of a web application, attackers can force the web server to spend all of its resources evaluating regular expressions thereby making the service inaccessible to genuine users.

This rule flags any execution of a hardcoded regular expression which has at least 3 characters and at least two instances of any of the following characters: `*+{`.

Example: `(a+)*`

Exceptions

Some corner-case regular expressions will not raise an issue even though they might be vulnerable. For example: `(a|aa)+`, `(a|a?)+`.

It is a good idea to test your regular expression if it has the same pattern on both side of a "`|`".

assess_the_problem

Ask Yourself Whether

- the executed regular expression is sensitive and a user can provide a string which will be analyzed by this regular expression.
- your regular expression engine performance decrease with specially crafted inputs and regular expressions.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const regex = /(a+)+b/; // Sensitive
const regex2 = new RegExp("(a+)+b"); // Sensitive

str.search("(a+)+b"); // Sensitive
str.match("(a+)+b"); // Sensitive
str.split("(a+)+b"); // Sensitive
```

Note: String.matchAll does not raise any issue as it is not supported by NodeJS.

how_to_fix

Recommended Secure Coding Practices

Check whether your regular expression engine (the algorithm executing your regular expression) has any known vulnerabilities. Search for vulnerability reports mentioning the one engine you’re are using.

Use if possible a library which is not vulnerable to Redos Attacks such as Google Re2.

Remember also that a ReDos attack is possible if a user-provided regular expression is executed. This rule won’t detect this kind of injection.

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-624 - Executable Regular Expression Error
- OWASP Regular expression Denial of Service - ReDoS

Using weak hashing algorithms is security-sensitive

Clave: javascript:S4790

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Cryptographic hash algorithms such as `MD2`, `MD4`, `MD5`, `MD6`, `HAVAL-128`, `HMAC-MD5`, `DSA` (which uses `SHA-1`), `RIPEMD`, `RIPEMD-128`, `RIPEMD-160`, `HMACRIPEMD160` and `SHA-1` are no longer considered secure, because it is possible to have `collisions` (little computational effort is enough to find two or more different inputs that produce the same hash).

default

Cryptographic hash algorithms such as `MD2`, `MD4`, `MD5`, `MD6`, `HAVAL-128`, `HMAC-MD5`, `DSA` (which uses `SHA-1`), `RIPEMD`, `RIPEMD-128`, `RIPEMD-160`, `HMACRIPEMD160` and `SHA-1` are no longer considered secure, because it is possible to have `collisions` (little computational effort is enough to find two or more different inputs that produce the same hash).

Ask Yourself Whether

The hashed value is used in a security context like:

- User-password storage.
- Security token generation (used to confirm e-mail when registering on a website, reset password, etc …).
- To compute some message integrity.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Safer alternatives, such as `SHA-256`, `SHA-512`, `SHA-3` are recommended, and for password hashing, it’s even better to use algorithms that do not compute too "quickly", like `bcrypt`, `scrypt`, `argon2` or `pbkdf2` because it slows down `brute force attacks`.

Sensitive Code Example

```
const crypto = require("crypto");

const hash = crypto.createHash('sha1'); // Sensitive
```

Compliant Solution

```
const crypto = require("crypto");

const hash = crypto.createHash('sha512'); // Compliant
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-1240 - Use of a Risky Cryptographic Primitive

assess_the_problem

Ask Yourself Whether

The hashed value is used in a security context like:

- User-password storage.
- Security token generation (used to confirm e-mail when registering on a website, reset password, etc …).
- To compute some message integrity.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const crypto = require("crypto");

const hash = crypto.createHash('sha1'); // Sensitive
```

how_to_fix

Recommended Secure Coding Practices

Safer alternatives, such as `SHA-256`, `SHA-512`, `SHA-3` are recommended, and for password hashing, it’s even better to use algorithms that do not compute too "quickly", like `bcrypt`, `scrypt`, `argon2` or `pbkdf2` because it slows down `brute force attacks`.

Compliant Solution

```
const crypto = require("crypto");

const hash = crypto.createHash('sha512'); // Compliant
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-1240 - Use of a Risky Cryptographic Primitive

Executing XPath expressions is security-sensitive

Clave: javascript:S4817

Severidad: CRITICAL

Estado: ⚠️ Depreciada

Descripción: No disponible

default

This rule is deprecated, and will eventually be removed.

Executing XPATH expressions is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2016-6272
- CVE-2016-9149
- CVE-2012-4837

User-provided data such as URL parameters should always be considered as untrusted and tainted. Constructing XPath expressions directly from tainted data enables attackers to inject specially crafted values that changes the initial meaning of the expression itself. Successful XPath injections attacks can read sensitive information from the XML document.

Ask Yourself Whether

- the XPATH expression might contain some unsafe input coming from a user.

You are at risk if you answered yes to this question.

Recommended Secure Coding Practices

Sanitize any user input before using it in an XPATH expression.

Sensitive Code Example

```
// === Server side ===

var xpath = require('xpath');
var xmldom = require('xmldom');

var doc = new xmldom.DOMParser().parseFromString(xml);
var nodes = xpath.select(userinput, doc); // Sensitive
var node = xpath.select1(userinput, doc); // Sensitive
```

```
// === Client side ===

// Chrome, Firefox, Edge, Opera, and Safari use the evaluate() method to select nodes:
var nodes = document.evaluate(userinput, xmlDoc, null, XPathResult.ANY_TYPE, null); // Sensitive

// Internet Explorer uses its own methods to select nodes:
var nodes = xmlDoc.selectNodes(userinput); // Sensitive
var node = xmlDoc.SelectSingleNode(userinput); // Sensitive
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-643 - Improper Neutralization of Data within XPath Expressions

assess_the_problem

Ask Yourself Whether

- the XPATH expression might contain some unsafe input coming from a user.

You are at risk if you answered yes to this question.

Sensitive Code Example

```
// === Server side ===

var xpath = require('xpath');
var xmldom = require('xmldom');

var doc = new xmldom.DOMParser().parseFromString(xml);
var nodes = xpath.select(userinput, doc); // Sensitive
var node = xpath.select1(userinput, doc); // Sensitive
```

```
// === Client side ===

// Chrome, Firefox, Edge, Opera, and Safari use the evaluate() method to select nodes:
var nodes = document.evaluate(userinput, xmlDoc, null, XPathResult.ANY_TYPE, null); // Sensitive

// Internet Explorer uses its own methods to select nodes:
var nodes = xmlDoc.selectNodes(userinput); // Sensitive
var node = xmlDoc.SelectSingleNode(userinput); // Sensitive
```

root_cause

This rule is deprecated, and will eventually be removed.

Executing XPATH expressions is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2016-6272
- CVE-2016-9149
- CVE-2012-4837

User-provided data such as URL parameters should always be considered as untrusted and tainted. Constructing XPath expressions directly from tainted data enables attackers to inject specially crafted values that changes the initial meaning of the expression itself. Successful XPath injections attacks can read sensitive information from the XML document.

how_to_fix

Recommended Secure Coding Practices

Sanitize any user input before using it in an XPATH expression.

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-643 - Improper Neutralization of Data within XPath Expressions

---

Using Sockets is security-sensitive

Clave: javascript:S4818

Severidad: CRITICAL

Estado: ⚠ Depreciada

Descripción: No disponible

default

This rule is deprecated, and will eventually be removed.

Using sockets is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2011-178
- CVE-2017-5645
- CVE-2018-6597

Sockets are vulnerable in multiple ways:

- They enable a software to interact with the outside world. As this world is full of attackers it is necessary to check that they cannot receive sensitive information or inject dangerous input.
- The number of sockets is limited and can be exhausted. Which makes the application unresponsive to users who need additional sockets.

This rules flags code that creates sockets. It matches only the direct use of sockets, not use through frameworks or high-level APIs such as the use of http connections.

Ask Yourself Whether

- sockets are created without any limit every time a user performs an action.
- input received from sockets is used without being sanitized.
- sensitive data is sent via sockets without being encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- In many cases there is no need to open a socket yourself. Use instead libraries and existing protocols.
- Encrypt all data sent if it is sensitive. Usually it is better to encrypt it even if the data is not sensitive as it might change later.
- Sanitize any input read from the socket.
- Limit the number of sockets a given user can create. Close the sockets as soon as possible.

Sensitive Code Example

```
const net = require('net');

var socket = new net.Socket(); // Sensitive
socket.connect(80, 'google.com');

// net.createConnection creates a new net.Socket, initiates connection with socket.connect(), then returns the net.Socket that starts the connection
net.createConnection({ port: port }, () => {}); // Sensitive
```

```
// net.connect is an alias to net.createConnection
net.connect({ port: port }, () => {}); // Sensitive
```

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-400 - Uncontrolled Resource Consumption ('Resource Exhaustion')
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

root_cause

This rule is deprecated, and will eventually be removed.

Using sockets is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2011-178
- CVE-2017-5645
- CVE-2018-6597

Sockets are vulnerable in multiple ways:

- They enable a software to interact with the outside world. As this world is full of attackers it is necessary to check that they cannot receive sensitive information or inject dangerous input.
- The number of sockets is limited and can be exhausted. Which makes the application unresponsive to users who need additional sockets.

This rules flags code that creates sockets. It matches only the direct use of sockets, not use through frameworks or high-level APIs such as the use of http connections.

assess_the_problem

Ask Yourself Whether

- sockets are created without any limit every time a user performs an action.
- input received from sockets is used without being sanitized.
- sensitive data is sent via sockets without being encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const net = require('net');

var socket = new net.Socket(); // Sensitive
socket.connect(80, 'google.com');

// net.createConnection creates a new net.Socket, initiates connection with socket.connect(), then returns the net.Socket that starts the connection
net.createConnection({ port: port }, () => {}); // Sensitive

// net.connect is an alias to net.createConnection
net.connect({ port: port }, () => {}); // Sensitive
```

how_to_fix

Recommended Secure Coding Practices

- In many cases there is no need to open a socket yourself. Use instead libraries and existing protocols.
- Encrypt all data sent if it is sensitive. Usually it is better to encrypt it even if the data is not sensitive as it might change later.
- Sanitize any input read from the socket.
- Limit the number of sockets a given user can create. Close the sockets as soon as possible.

See

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-400 - Uncontrolled Resource Consumption ('Resource Exhaustion')
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

---

Promise rejections should not be caught by "try" blocks

Clave: javascript:S4822

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

An exception thrown inside a promise will not be caught by a nesting `try` block due to the asynchronous nature of execution.

Promises are designed to propagate errors to the next error handler or catch() block in the promise chain. Promises are asynchronous and operate outside of the normal call stack. When a Promise is created, it is added to the microtask queue, which is processed after the current call stack has completed. This means that the try-catch block surrounding the Promise will have already completed by the time the Promise is resolved or rejected. Therefore, any error occurring within the Promise will not be caught by the try-catch block.

```
function foo() {
  try { // Noncompliant: Promise rejection will not be caught
    runPromiseThatRejects();
  } catch (e) {
    console.log("Failed to run promise", e);
  }
}
```

Instead of using a try-catch block to handle errors in a Promise chain, use the Promise.catch() method. This method allows you to specify a callback function that will be executed if the Promise is rejected.

```
function foo() {
  runPromiseThatRejects().catch(e => console.log("Failed to run promise", e));
}
```

Alternatively, wait for the Promise fulfillment value using `await`. It is used to unwrap promises and pauses the execution of its surrounding `async` function until the promise is settled (that is, fulfilled or rejected). Any errors that occur within the Promise will be thrown as exceptions.

```
async function foo() {
  try {
    await runPromiseThatRejects();
  } catch (e) {
    console.log("Failed to run promise", e);
  }
}
```

This rule reports `try...catch` statements containing nothing else but call(s) to a function returning a `Promise` (thus, it’s less likely that `catch` is intended to catch something else than `Promise` rejection).

resources

Documentation

- MDN web docs - Promise
- MDN web docs - `try...catch`
- MDN web docs - `await`
- MDN web docs - `async function`
- MDN web docs - Microtasks

Using command line arguments is security-sensitive

Clave: javascript:S4823

Severidad: CRITICAL

Estado: ⚠️ Depreciada

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- any of the command line arguments are used without being sanitized first.
- your application accepts sensitive information via command line arguments.

If you answered yes to any of these questions you are at risk.

Sensitive Code Example

```
// The process object is a global that provides information about, and control over, the current Node.js process
var param = process.argv[2]; // Sensitive: check how the argument is used
console.log('Param: ' + param);
```

root_cause

This rule is deprecated, and will eventually be removed.

Using command line arguments is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-7281
- CVE-2018-12326
- CVE-2011-3198

Command line arguments can be dangerous just like any other user input. They should never be used without being first validated and sanitized.

Remember also that any user can retrieve the list of processes running on a system, which makes the arguments provided to them visible. Thus passing sensitive information via command line arguments should be considered as insecure.

This rule raises an issue when on every program entry points (`main` methods) when command line arguments are used. The goal is to guide security code reviews.

how_to_fix

Recommended Secure Coding Practices

Sanitize all command line arguments before using them.

Any user or application can list running processes and see the command line arguments they were started with. There are safer ways of providing sensitive information to an application than exposing them in the command line. It is common to write them on the process' standard input, or give the path to a file containing the information.

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-88 - Argument Injection or Modification
- CWE - CWE-214 - Information Exposure Through Process Environment

default

This rule is deprecated, and will eventually be removed.

Using command line arguments is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-7281
- CVE-2018-12326
- CVE-2011-3198

Command line arguments can be dangerous just like any other user input. They should never be used without being first validated and sanitized.

Remember also that any user can retrieve the list of processes running on a system, which makes the arguments provided to them visible. Thus passing sensitive information via command line arguments should be considered as insecure.

This rule raises an issue when on every program entry points (`main` methods) when command line arguments are used. The goal is to guide security code reviews.

Ask Yourself Whether

- any of the command line arguments are used without being sanitized first.
- your application accepts sensitive information via command line arguments.

If you answered yes to any of these questions you are at risk.

Recommended Secure Coding Practices

Sanitize all command line arguments before using them.

Any user or application can list running processes and see the command line arguments they were started with. There are safer ways of providing sensitive information to an application than exposing them in the command line. It is common to write them on the process' standard input, or give the path to a file containing the information.

Sensitive Code Example

```
// The process object is a global that provides information about, and control over, the current Node.js process
var param = process.argv[2]; // Sensitive: check how the argument is used
console.log('Param: ' + param);
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-88 - Argument Injection or Modification
- CWE - CWE-214 - Information Exposure Through Process Environment

Reading the Standard Input is security-sensitive

Clave: javascript:S4829

Severidad: CRITICAL

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule is deprecated, and will eventually be removed.

Reading Standard Input is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2005-2337
- CVE-2017-11449

It is common for attackers to craft inputs enabling them to exploit software vulnerabilities. Thus any data read from the standard input (stdin) can be dangerous and should be validated.

This rule flags code that reads from the standard input.

how_to_fix

Recommended Secure Coding Practices

Sanitize all data read from the standard input before using it.

See

- CWE - CWE-20 - Improper Input Validation

default

This rule is deprecated, and will eventually be removed.

Reading Standard Input is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2005-2337
- CVE-2017-11449

It is common for attackers to craft inputs enabling them to exploit software vulnerabilities. Thus any data read from the standard input (stdin) can be dangerous and should be validated.

This rule flags code that reads from the standard input.

Ask Yourself Whether

- data read from the standard input is not sanitized before being used.

You are at risk if you answered yes to this question.

Recommended Secure Coding Practices

Sanitize all data read from the standard input before using it.

Sensitive Code Example

```
// The process object is a global that provides information about, and control over, the current Node.js process
// All uses of process.stdin are security-sensitive and should be reviewed

process.stdin.on('readable', () => {
        const chunk = process.stdin.read(); // Sensitive
        if (chunk !== null) {
                dosomething(chunk);
        }
});

const readline = require('readline');
readline.createInterface({
        input: process.stdin // Sensitive
}).on('line', (input) => {
        dosomething(input);
});
```

See

- CWE - CWE-20 - Improper Input Validation

assess_the_problem

Ask Yourself Whether

- data read from the standard input is not sanitized before being used.

You are at risk if you answered yes to this question.

Sensitive Code Example

```
// The process object is a global that provides information about, and control over, the current Node.js process
// All uses of process.stdin are security-sensitive and should be reviewed

process.stdin.on('readable', () => {
	const chunk = process.stdin.read(); // Sensitive
	if (chunk !== null) {
		dosomething(chunk);
	}
});

const readline = require('readline');
readline.createInterface({
	input: process.stdin // Sensitive
}).on('line', (input) => {
	dosomething(input);
});
```

Server certificates should be verified during SSL/TLS connections

Clave: javascript:S4830

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

introduction

This vulnerability makes it possible that an encrypted communication is intercepted.

resources

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-295 - Improper Certificate Validation
- STIG Viewer - Application Security and Development: V-222550 - The application must validate certificates by constructing a certification path to an accepted trust anchor.

how_to_fix

The following code contains examples of disabled certificate validation.

The certificate validation gets disabled by setting `rejectUnauthorized` to `false`. To enable validation set the value to `true` or do not set `rejectUnauthorized` at all to use the secure default value.

Noncompliant code example

```
const https = require('node:https');

let options = {
  hostname: 'www.example.com',
  port: 443,
  path: '/',
  method: 'GET',
  rejectUnauthorized: false,
  secureProtocol: 'TLSv1_2_method'
};
```

```
let req = https.request(options, (res) => {
  res.on('data', (d) => {
    process.stdout.write(d);
  });
}); // Noncompliant

const tls = require('node:tls');

let options = {
    rejectUnauthorized: false,
    secureProtocol: 'TLSv1_2_method'
};

let socket = tls.connect(443, "www.example.com", options, () => {
  process.stdin.pipe(socket);
  process.stdin.resume();
}); // Noncompliant
```

Compliant solution

```
const https = require('node:https');

let options = {
  hostname: 'www.example.com',
  port: 443,
  path: '/',
  method: 'GET',
  secureProtocol: 'TLSv1_2_method'
};

let req = https.request(options, (res) => {
  res.on('data', (d) => {
    process.stdout.write(d);
  });
});

const tls = require('node:tls');

let options = {
    secureProtocol: 'TLSv1_2_method'
};

let socket = tls.connect(443, "www.example.com", options, () => {
  process.stdin.pipe(socket);
  process.stdin.resume();
});
```

How does this work?

Addressing the vulnerability of disabled TLS certificate validation primarily involves re-enabling the default validation.

To avoid running into problems with invalid certificates, consider the following sections.

Using trusted certificates

If possible, always use a certificate issued by a well-known, trusted CA for your server. Most programming environments come with a predefined list of trusted root CAs, and certificates issued by these authorities are validated automatically. This is the best practice, and it requires no additional code or configuration.

Working with self-signed certificates or non-standard CAs

In some cases, you might need to work with a server using a self-signed certificate, or a certificate issued by a CA not included in your trusted roots. Rather than disabling certificate validation in your code, you can add the necessary certificates to your trust store.

root_cause

Transport Layer Security (TLS) provides secure communication between systems over the internet by encrypting the data sent between them. Certificate validation adds an extra layer of trust and security to this process to ensure that a system is indeed the one it claims to be.

When certificate validation is disabled, the client skips a critical security check. This creates an opportunity for attackers to pose as a trusted entity and intercept, manipulate, or steal the data being transmitted.

What is the potential impact?

Establishing trust in a secure way is a non-trivial task. When you disable certificate validation, you are removing a key mechanism designed to build this trust in internet communication, opening your system up to a number of potential threats.

Identity spoofing

If a system does not validate certificates, it cannot confirm the identity of the other party involved in the communication. An attacker can exploit this by creating a fake server and masquerading as a legitimate one. For example, they might set up a server that looks like your bank’s server, tricking your system into thinking it is communicating with the bank. This scenario, called identity spoofing, allows the attacker to collect any data your system sends to them, potentially leading to significant data breaches.

Loss of data integrity

When TLS certificate validation is disabled, the integrity of the data you send and receive cannot be guaranteed. An attacker could modify the data in transit, and you would have no way of knowing. This could range from subtle manipulations of the data you receive to the injection of malicious code or malware into your system. The consequences of such breaches of data integrity can be severe, depending on the nature of the data and the system.

how_to_fix

The following code contains examples of disabled certificate validation.

The certificate validation gets disabled by setting `rejectUnauthorized` to `false`. To enable validation set the value to `true` or do not set `rejectUnauthorized` at all to use the secure default value.

Noncompliant code example

```
const request = require('request');

let socket = request.get({
  url: 'www.example.com',
  rejectUnauthorized: false, // Noncompliant
  secureProtocol: 'TLSv1_2_method'
});
```

Compliant solution

```
const request = require('request');

let socket = request.get({
  url: 'https://www.example.com/',
  secureProtocol: 'TLSv1_2_method'
});
```

How does this work?

Addressing the vulnerability of disabled TLS certificate validation primarily involves re-enabling the default validation.

To avoid running into problems with invalid certificates, consider the following sections.

Using trusted certificates

If possible, always use a certificate issued by a well-known, trusted CA for your server. Most programming environments come with a predefined list of trusted root CAs, and certificates issued by these authorities are validated automatically. This is the best practice, and it requires no additional code or configuration.

Working with self-signed certificates or non-standard CAs

In some cases, you might need to work with a server using a self-signed certificate, or a certificate issued by a CA not included in your trusted roots. Rather than disabling certificate validation in your code, you can add the necessary certificates to your trust store.

Server hostnames should be verified during SSL/TLS connections

Clave: javascript:S5527

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-297 - Improper Validation of Certificate with Host Mismatch
- STIG Viewer - Application Security and Development: V-222550 - The application must validate certificates by constructing a certification path to an accepted trust anchor.

how_to_fix

The following code contains examples of disabled hostname validation.

The hostname validation gets disabled by overriding `checkServerIdentity` with an empty implementation. It is highly recommended to use the original implementation.

Noncompliant code example

```
const https = require('node:https');

let options = {
  hostname: 'www.example.com',
  port: 443,
  path: '/',
  method: 'GET',
  checkServerIdentity: function() {}, // Noncompliant
  secureProtocol: 'TLSv1_2_method'
};

let req = https.request(options, (res) => {
  res.on('data', (d) => {
    process.stdout.write(d);
  });
});
```

```
const tls = require('node:tls');

let options = {
  checkServerIdentity: function() {}, // Noncompliant
  secureProtocol: 'TLSv1_2_method'
};

let socket = tls.connect(443, "www.example.com", options, () => {
  process.stdin.pipe(socket);
  process.stdin.resume();
});
```

Compliant solution

```
const https = require('node:https');

let options = {
  hostname: 'www.example.com',
  port: 443,
  path: '/',
  method: 'GET',
  secureProtocol: 'TLSv1_2_method'
};

let req = https.request(options, (res) => {
  res.on('data', (d) => {
    process.stdout.write(d);
  });
});
```

```
const tls = require('node:tls');

let options = {
  secureProtocol: 'TLSv1_2_method'
};

let socket = tls.connect(443, "www.example.com", options, () => {
  process.stdin.pipe(socket);
  process.stdin.resume();
});
```

How does this work?

To fix the vulnerability of disabled hostname validation, it is strongly recommended to first re-enable the default validation and fix the root cause: the validity of the certificate.

Use valid certificates

If a hostname validation failure prevents connecting to the target server, keep in mind that **one system's code should not work around another system's problems**, as this creates unnecessary dependencies and can lead to reliability issues.

Therefore, the first solution is to change the remote host's certificate to match its identity. If the remote host is not under your control, consider replicating its service to a server whose certificate you can change yourself.

In case the contacted host is located on a development machine, and if there is no other choice, try following this solution:

- Create a self-signed certificate for that machine.
- Add this self-signed certificate to the system's trust store.
- If the hostname is not `localhost`, add the hostname in the `/etc/hosts` file.

introduction

This vulnerability allows attackers to impersonate a trusted host.

root_cause

Transport Layer Security (TLS) provides secure communication between systems over the internet by encrypting the data sent between them. In this process, the role of hostname validation, combined with certificate validation, is to ensure that a system is indeed the one it claims to be, adding an extra layer of trust and security.

When hostname validation is disabled, the client skips this critical check. This creates an opportunity for attackers to pose as a trusted entity and intercept, manipulate, or steal the data being transmitted.

To do so, an attacker would obtain a valid certificate authenticating `example.com`, serve it using a different hostname, and the application code would still accept it.

What is the potential impact?

Establishing trust in a secure way is a non-trivial task. When you disable hostname validation, you are removing a key mechanism designed to build this trust in internet communication, opening your system up to a number of potential threats.

Identity spoofing

If a system does not validate hostnames, it cannot confirm the identity of the other party involved in the communication. An attacker can exploit this by creating a fake server and masquerading it as a legitimate one. For example, they might set up a server that looks like your bank’s server, tricking your system into thinking it is communicating with the bank. This scenario, called identity spoofing, allows the attacker to collect any data your system sends to them, potentially leading to significant data breaches.

Forwarding client IP address is security-sensitive

Clave: javascript:S5759

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Users often connect to web servers through HTTP proxies.

Proxy can be configured to forward the client IP address via the `X-Forwarded-For` or `Forwarded` HTTP headers.

IP address is a personal information which can identify a single user and thus impact his privacy.

assess_the_problem

Ask Yourself Whether

- The web application uses reverse proxies or similar but doesn’t need to know the IP address of the user.

There is a risk if you answered yes to this question.

Sensitive Code Example

node-http-proxy

```
var httpProxy = require('http-proxy');

httpProxy.createProxyServer({target:'http://localhost:9000', xfwd:true}) // Noncompliant
  .listen(8000);
```

http-proxy-middleware

```
var express = require('express');

const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

app.use('/proxy', createProxyMiddleware({ target: 'http://localhost:9000', changeOrigin: true, xfwd: true })); // Noncompliant
app.listen(3000);
```

default

Users often connect to web servers through HTTP proxies.

Proxy can be configured to forward the client IP address via the `X-Forwarded-For` or `Forwarded` HTTP headers.

IP address is a personal information which can identify a single user and thus impact his privacy.

Ask Yourself Whether

- The web application uses reverse proxies or similar but doesn’t need to know the IP address of the user.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

User IP address should not be forwarded unless the application needs it, as part of an authentication, authorization scheme or log management for examples.

Sensitive Code Example

node-http-proxy

```
var httpProxy = require('http-proxy');

httpProxy.createProxyServer({target:'http://localhost:9000', xfwd:true}) // Noncompliant
  .listen(8000);
```

http-proxy-middleware

```
var express = require('express');

const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

app.use('/proxy', createProxyMiddleware({ target: 'http://localhost:9000', changeOrigin: true, xfwd: true })); // Noncompliant
app.listen(3000);
```

Compliant Solution

node-http-proxy

```
var httpProxy = require('http-proxy');

// By default xfwd option is false
httpProxy.createProxyServer({target:'http://localhost:9000'}) // Compliant
  .listen(8000);
```

http-proxy-middleware

```
var express = require('express');

const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

// By default xfwd option is false
app.use('/proxy', createProxyMiddleware({ target: 'http://localhost:9000', changeOrigin: true})); // Compliant
app.listen(3000);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - X-Forwarded-For

how_to_fix

Recommended Secure Coding Practices

User IP address should not be forwarded unless the application needs it, as part of an authentication, authorization scheme or log management for examples.

Compliant Solution

node-http-proxy

```
var httpProxy = require('http-proxy');

// By default xfwd option is false
httpProxy.createProxyServer({target:'http://localhost:9000'}) // Compliant
  .listen(8000);
```

http-proxy-middleware

```
var express = require('express');
```

```
const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

// By default xfwd option is false
app.use('/proxy', createProxyMiddleware({ target: 'http://localhost:9000', changeOrigin: true})); // Compliant
app.listen(3000);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- developer.mozilla.org - X-Forwarded-For

Single-character alternations in regular expressions should be replaced with character classes

Clave: javascript:S6035

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When an alternation contains multiple alternatives that consist of a single character, it can be rewritten as a character class. This should be preferred because it is more efficient and can even help prevent stack overflows when used inside a repetition (see rule S5998).

Noncompliant code example

```
/a|b|c/; // Noncompliant
```

Compliant solution

```
/[abc]/;
// or
/[a-c]/;
```

Disabling Mocha timeouts should be explicit

Clave: javascript:S6080

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Mocha Documentation - Timeouts
- Mocha Documentation - Disabling timeouts
- MDN web docs - Maximum_delay_value

root_cause

Setting timeouts with Mocha allows you to control the maximum time a test case or suite can take to execute. However, incorrect usage or excessive timeout values can lead to undesired consequences and impact the effectiveness of your test suite. For example, setting a timeout by calling `this.timeout()` with a value greater than the maximum delay (2,147,483,647 ms) will cause the timeout to be disabled.

```
describe("testing this.timeout", function() {
  it("unexpectedly disables the timeout", function(done) {
    this.timeout(2147483648); // Noncompliant: the timeout is disabled
  });
});
```

When using `this.timeout()`, make sure to set a reasonable value that allows your tests to complete within a reasonable timeframe.

```
describe("testing this.timeout", function() {
  it("sets the timeout to 1'000 milliseconds", function(done) {
    this.timeout(1000);
  });
});
```

If the goal is really to disable the timeout, set it to zero instead.

```
describe("testing this.timeout", function() {
  it("disables the timeout as expected", function(done) {
    this.timeout(0);
  });
});
```

Chai assertions should have only one reason to succeed

Clave: javascript:S6092

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A unit test assertion should have only one reason to succeed because it helps to ensure that the test is focused and specific. When a test has multiple reasons to succeed, it becomes difficult to determine the root cause of a failure if the test fails. This can lead to confusion and wasted time trying to debug the test.

This rule raises an issue when the following Chai.js assertions are found:

- When `.not` and `.throw` are used together and at least one argument is provided to `.throw`. Such assertions succeed when the target either does not throw any exception, or when it throws an exception different from the one provided.
- When `.not` and `.include` are used together and an `object` is given to `.include`. Such assertions succeed when one or multiple key/values are missing.
- When `.not` and `.property` are used together and `.property` is given at least two arguments. Such assertions succeed when the target either doesn’t have the requested property, or when this property exists but has a different value.
- When `.not` and `.ownPropertyDescriptor` are used together and `.ownPropertyDescriptor` is given at least two arguments. Such assertions succeed when the target either doesn’t have the requested property descriptor, or its property descriptor is not deeply equal to the given descriptor
- When `.not` and `.members` are used together. Such assertions succeed when one or multiple members are missing.
- When `.change` and `.by` are used together. Such assertions succeed when the target either decreases or increases by the given delta
- When `.not` and `.increase` are used together. Such assertions succeed when the target either decreases or stays the same.
- When `.not` and `.decrease` are used together. Such assertions succeed when the target either increases or stays the same.
- When `.not` negates `.by`. Such assertions succeed when the target didn’t change by one specific delta among all the possible deltas.
- When `.not` and `.finite` are used together. Such assertions succeed when the target either is not a `number`, or is one of `Nan`, positive `Infinity`, negative `Infinity`.

```
const expect = require('chai').expect;

describe("Each Chai.js assertion", function() {
    const throwsTypeError = () => { throw new TypeError() }

    it("has more than one reason to succeed", function() {
        expect(throwsTypeError).to.not.throw(ReferenceError) // Noncompliant
        expect({a: 42}).to.not.include({b: 10, c: 20});  // Noncompliant
        expect({a: 21}).to.not.have.property('b', 42); // Noncompliant
        expect({a: 21}).to.not.have.ownPropertyDescriptor('b', {   // Noncompliant
            configurable: true,
            enumerable: true,
            writable: true,
            value: 42,
        });
        expect([21, 42]).to.not.have.members([1, 2]); // Noncompliant

        let myObj = { value: 1 }
        const incThree = () => { myObj.value += 3; };
        const decThree = () => { myObj.value -= 3; };
        const doNothing = () => {};

        expect(incThree).to.change(myObj, 'value').by(3); // Noncompliant
        expect(decThree).to.change(myObj, 'value').by(3); // Noncompliant

        expect(decThree).to.not.increase(myObj, 'value'); // Noncompliant
        expect(incThree).to.not.decrease(myObj, 'value'); // Noncompliant

        expect(doNothing).to.not.increase(myObj, 'value'); // Noncompliant
        expect(doNothing).to.not.decrease(myObj, 'value'); // Noncompliant

        expect(incThree).to.increase(myObj, 'value').but.not.by(1); // Noncompliant

        let toCheck;
        expect(toCheck).to.not.be.finite; // Noncompliant
    });
});
```

By having only one reason to succeed, the test is more precise and easier to understand. It also helps to ensure that the test is testing only one specific behavior or functionality of the code, which makes it easier to identify and fix any issues that arise.

```
const expect = require('chai').expect;
```

```
describe("Each Chai.js assertion", function() {
    const throwsTypeError = () => { throw new TypeError() }

    it("has only one reason to succeed", function() {
        expect(throwsTypeError).to.throw(TypeError)
        expect({a: 42}).to.not.have.any.keys('b', 'c');
        expect({a: 21}).to.not.have.property('b');
        expect({a: 21}).to.not.have.ownPropertyDescriptor('b');
        expect([21, 42]).to.not.include(1).and.not.include(2);

        let myObj = { value: 1 }
        const incThree = () => { myObj.value += 3; };
        const decThree = () => { myObj.value -= 3; };
        const doNothing = () => {};

        expect(incThree).to.increase(myObj, 'value').by(3);
        expect(decThree).to.decrease(myObj, 'value').by(3);

        expect(decThree).to.decrease(myObj, 'value').by(3);
        expect(incThree).to.increase(myObj, 'value').by(3);

        expect(doNothing).to.not.change(myObj, 'value');

        expect(incThree).to.increase(myObj, 'value').by(3);

        let toCheck;
        // Either of the following is valid
        expect(toCheck).to.be.a('string');
        expect(toCheck).to.be.NaN;
        expect(toCheck).to.equal(Infinity);
        expect(toCheck).to.equal(-Infinity);
    });
});
```

Having only one reason to succeed also helps to make the test more maintainable. If the test needs to be updated or modified in the future, it is easier to do so when the test is focused on a single behavior or functionality.

resources

Documentation

- Chai.js Documentation - `.by`
- Chai.js Documentation - `.change`
- Chai.js Documentation - `.decrease`
- Chai.js Documentation - `.finite`
- Chai.js Documentation - `.include`
- Chai.js Documentation - `.increase`
- Chai.js Documentation - `.members`
- Chai.js Documentation - `.ownPropertyDescriptor`
- Chai.js Documentation - `.property`
- Chai.js Documentation - `.throw`

Granting access to S3 buckets to all or authenticated users is security-sensitive

Clave: javascript:S6265

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

default

Predefined permissions, also known as canned ACLs, are an easy way to grant large privileges to predefined groups or users.

The following canned ACLs are security-sensitive:

- `PUBLIC_READ`, `PUBLIC_READ_WRITE` grant respectively "read" and "read and write" privileges to anyone, either authenticated or anonymous (`AllUsers` group).
- `AUTHENTICATED_READ` grants "read" privilege to all authenticated users (`AuthenticatedUsers` group).

Ask Yourself Whether

- The S3 bucket stores sensitive data.
- The S3 bucket is not used to store static resources of websites (images, css …).

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to implement the least privilege policy, i.e., to only grant users the necessary permissions for their required tasks. In the context of canned ACL, set it to `PRIVATE` (the default one), and if needed more granularity then use an appropriate S3 policy.

Sensitive Code Example

All users, either authenticated or anonymous, have read and write permissions with the `PUBLIC_READ_WRITE` access control:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'bucket', {
    accessControl: s3.BucketAccessControl.PUBLIC_READ_WRITE // Sensitive
});

new s3deploy.BucketDeployment(this, 'DeployWebsite', {
    accessControl: s3.BucketAccessControl.PUBLIC_READ_WRITE // Sensitive
});
```

Compliant Solution

With the `PRIVATE` access control (default), only the bucket owner has the read/write permissions on the bucket and its ACL.

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'bucket', {
    accessControl: s3.BucketAccessControl.PRIVATE
});

new s3deploy.BucketDeployment(this, 'DeployWebsite', {
    accessControl: s3.BucketAccessControl.PRIVATE
});
```

See

- AWS Documentation - Access control list (ACL) overview (canned ACLs)
- AWS Documentation - Controlling access to a bucket with user policies
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control
- AWS CDK version 2 - Class Bucket (construct)

root_cause

Predefined permissions, also known as canned ACLs, are an easy way to grant large privileges to predefined groups or users.

The following canned ACLs are security-sensitive:

- `PUBLIC_READ`, `PUBLIC_READ_WRITE` grant respectively "read" and "read and write" privileges to anyone, either authenticated or anonymous (`AllUsers` group).
- `AUTHENTICATED_READ` grants "read" privilege to all authenticated users (`AuthenticatedUsers` group).

how_to_fix

Recommended Secure Coding Practices

It’s recommended to implement the least privilege policy, i.e., to only grant users the necessary permissions for their required tasks. In the context of canned ACL, set it to `PRIVATE` (the default one), and if needed more granularity then use an appropriate S3 policy.

Compliant Solution

With the `PRIVATE` access control (default), only the bucket owner has the read/write permissions on the bucket and its ACL.

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'bucket', {
    accessControl: s3.BucketAccessControl.PRIVATE
});

new s3deploy.BucketDeployment(this, 'DeployWebsite', {
    accessControl: s3.BucketAccessControl.PRIVATE
});
```

See

- AWS Documentation - Access control list (ACL) overview (canned ACLs)
- AWS Documentation - Controlling access to a bucket with user policies
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control
- AWS CDK version 2 - Class Bucket (construct)

assess_the_problem

Ask Yourself Whether

- The S3 bucket stores sensitive data.
- The S3 bucket is not used to store static resources of websites (images, css …).

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

All users, either authenticated or anonymous, have read and write permissions with the `PUBLIC_READ_WRITE` access control:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'bucket', {
    accessControl: s3.BucketAccessControl.PUBLIC_READ_WRITE // Sensitive
});

new s3deploy.BucketDeployment(this, 'DeployWebsite', {
    accessControl: s3.BucketAccessControl.PUBLIC_READ_WRITE // Sensitive
});
```

Disabling Angular built-in sanitization is security-sensitive

Clave: javascript:S6268

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

default

Angular prevents XSS vulnerabilities by treating all values as untrusted by default. Untrusted values are systematically sanitized by the framework before they are inserted into the DOM.

Still, developers have the ability to manually mark a value as trusted if they are sure that the value is already sanitized. Accidentally trusting malicious data will introduce an XSS vulnerability in the application and enable a wide range of serious attacks like accessing/modifying sensitive information or impersonating other users.

Ask Yourself Whether

- The value for which sanitization has been disabled is user-controlled.
- It's difficult to understand how this value is constructed.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Avoid including dynamic executable code and thus disabling Angular's built-in sanitization unless it's absolutely necessary. Try instead to rely as much as possible on static templates and Angular built-in sanitization to define web page content.
- Make sure to understand how the value to consider as trusted is constructed and never concatenate it with user-controlled data.
- Make sure to choose the correct DomSanitizer "bypass" method based on the context. For instance, only use `bypassSecurityTrustUrl` to trust urls in an `href` attribute context.

Sensitive Code Example

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer, SafeHtml } from "@angular/platform-browser";
import { ActivatedRoute } from '@angular/router';

@Component({
  template: '<div id="hello" [innerHTML]="hello"></div>'
})
export class HelloComponent implements OnInit {
  hello: SafeHtml;

  constructor(private sanitizer: DomSanitizer, private route: ActivatedRoute) { }

  ngOnInit(): void {
    let name = this.route.snapshot.queryParams.name;
    let html = "<h1>Hello " + name + "</h1>";
    this.hello = this.sanitizer.bypassSecurityTrustHtml(html); // Sensitive
  }
}
```

Compliant Solution

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer } from "@angular/platform-browser";
import { ActivatedRoute } from '@angular/router';

@Component({
  template: '<div id="hello"><h1>Hello {{name}}</h1></div>',
})
export class HelloComponent implements OnInit {
  name: string;

  constructor(private sanitizer: DomSanitizer, private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.name = this.route.snapshot.queryParams.name;
  }
}
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Angular - Best Practices - Security

root_cause

Angular prevents XSS vulnerabilities by treating all values as untrusted by default. Untrusted values are systematically sanitized by the framework before they are inserted into the DOM.

Still, developers have the ability to manually mark a value as trusted if they are sure that the value is already sanitized. Accidentally trusting malicious data will introduce an XSS vulnerability in the application and enable a wide range of serious attacks like accessing/modifying sensitive information or impersonating other users.

how_to_fix

Recommended Secure Coding Practices

- Avoid including dynamic executable code and thus disabling Angular’s built-in sanitization unless it’s absolutely necessary. Try instead to rely as much as possible on static templates and Angular built-in sanitization to define web page content.
- Make sure to understand how the value to consider as trusted is constructed and never concatenate it with user-controlled data.
- Make sure to choose the correct DomSanitizer "bypass" method based on the context. For instance, only use `bypassSecurityTrustUrl` to trust urls in an `href` attribute context.

Compliant Solution

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer } from "@angular/platform-browser";
import { ActivatedRoute } from '@angular/router';

@Component({
  template: '<div id="hello"><h1>Hello {{name}}</h1></div>',
})
export class HelloComponent implements OnInit {
  name: string;

  constructor(private sanitizer: DomSanitizer, private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.name = this.route.snapshot.queryParams.name;
  }
}
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Angular - Best Practices - Security

assess_the_problem

Ask Yourself Whether

- The value for which sanitization has been disabled is user-controlled.
- It’s difficult to understand how this value is constructed.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer, SafeHtml } from "@angular/platform-browser";
import { ActivatedRoute } from '@angular/router';

@Component({
  template: '<div id="hello" [innerHTML]="hello"></div>'
})
export class HelloComponent implements OnInit {
  hello: SafeHtml;

  constructor(private sanitizer: DomSanitizer, private route: ActivatedRoute) { }

  ngOnInit(): void {
    let name = this.route.snapshot.queryParams.name;
    let html = "<h1>Hello " + name + "</h1>";
    this.hello = this.sanitizer.bypassSecurityTrustHtml(html); // Sensitive
  }
}
```

Policies authorizing public access to resources are security-sensitive

Clave: javascript:S6270

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The AWS resource stores or processes sensitive data.
- The AWS resource is designed to be private.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

This policy allows all users, including anonymous ones, to access an S3 bucket:

```
import { aws_iam as iam } from 'aws-cdk-lib'
import { aws_s3 as s3 } from 'aws-cdk-lib'

const bucket = new s3.Bucket(this, "ExampleBucket")

bucket.addToResourcePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["s3:*"],
    resources: [bucket.arnForObjects("*")],
    principals: [new iam.AnyPrincipal()] // Sensitive
}))
```

how_to_fix

Recommended Secure Coding Practices

It’s recommended to implement the least privilege principle, i.e. to grant necessary permissions only to users for their required tasks. In the context of resource-based policies, list the principals that need the access and grant to them only the required privileges.

Compliant Solution

This policy allows only the authorized users:

```
import { aws_iam as iam } from 'aws-cdk-lib'
import { aws_s3 as s3 } from 'aws-cdk-lib'

const bucket = new s3.Bucket(this, "ExampleBucket")

bucket.addToResourcePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["s3:*"],
    resources: [bucket.arnForObjects("*")],
    principals: [new iam.AccountRootPrincipal()]
}))
```

See

- AWS Documentation - Grant least privilege
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control
- STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.

root_cause

Resource-based policies granting access to all users can lead to information leakage.

default

Resource-based policies granting access to all users can lead to information leakage.

Ask Yourself Whether

- The AWS resource stores or processes sensitive data.
- The AWS resource is designed to be private.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to implement the least privilege principle, i.e. to grant necessary permissions only to users for their required tasks. In the context of resource-based policies, list the principals that need the access and grant to them only the required privileges.

Sensitive Code Example

This policy allows all users, including anonymous ones, to access an S3 bucket:

```
import { aws_iam as iam } from 'aws-cdk-lib'
import { aws_s3 as s3 } from 'aws-cdk-lib'

const bucket = new s3.Bucket(this, "ExampleBucket")

bucket.addToResourcePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["s3:*"],
    resources: [bucket.arnForObjects("*")],
    principals: [new iam.AnyPrincipal()] // Sensitive
}))
```

Compliant Solution

This policy allows only the authorized users:

```
import { aws_iam as iam } from 'aws-cdk-lib'
import { aws_s3 as s3 } from 'aws-cdk-lib'

const bucket = new s3.Bucket(this, "ExampleBucket")

bucket.addToResourcePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["s3:*"],
    resources: [bucket.arnForObjects("*")],
    principals: [new iam.AccountRootPrincipal()]
}))
```

See

- AWS Documentation - Grant least privilege
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control
- STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.

Using unencrypted EBS volumes is security-sensitive

Clave: javascript:S6275

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Amazon Elastic Block Store (EBS) is a block-storage service for Amazon Elastic Compute Cloud (EC2). EBS volumes can be encrypted, ensuring the security of both data-at-rest and data-in-transit between an instance and its attached EBS storage. In the case that adversaries gain physical access to the storage medium they are not able to access the data. Encryption can be enabled for specific volumes or for all new volumes and snapshots. Volumes created from snapshots inherit their encryption configuration. A volume created from an encrypted snapshot will also be encrypted by default.

Ask Yourself Whether

- The disk contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to encrypt EBS volumes that contain sensitive information. Encryption and decryption are handled transparently by EC2, so no further modifications to the application are necessary. Instead of enabling encryption for every volume, it is also possible to enable encryption globally for a specific region. While creating volumes from encrypted snapshots will result in them being encrypted, explicitly enabling this security parameter will prevent any future unexpected security downgrade.

Sensitive Code Example

For aws_cdk.aws_ec2.Volume:

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';

new Volume(this, 'unencrypted-explicit', {
      availabilityZone: 'us-west-2a',
      size: Size.gibibytes(1),
      encrypted: false // Sensitive
    });
```

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';

new Volume(this, 'unencrypted-implicit', {
      availabilityZone: 'eu-west-1a',
      size: Size.gibibytes(1),
    }); // Sensitive as encryption is disabled by default
```

Compliant Solution

For aws_cdk.aws_ec2.Volume:

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';

new Volume(this, 'encrypted-explicit', {
      availabilityZone: 'eu-west-1a',
      size: Size.gibibytes(1),
      encrypted: true
    });
```

See

- Amazon EBS encryption
- CWE - CWE-311 - Missing Encryption of Sensitive Data

how_to_fix

Recommended Secure Coding Practices

It’s recommended to encrypt EBS volumes that contain sensitive information. Encryption and decryption are handled transparently by EC2, so no further modifications to the application are necessary. Instead of enabling encryption for every volume, it is also possible to enable encryption globally for a specific region. While creating volumes from encrypted snapshots will result in them being encrypted, explicitly enabling this security parameter will prevent any future unexpected security downgrade.

Compliant Solution

For aws_cdk.aws_ec2.Volume:

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';
```

```
new Volume(this, 'encrypted-explicit', {
      availabilityZone: 'eu-west-1a',
      size: Size.gibibytes(1),
      encrypted: true
    });
```

See

- Amazon EBS encryption
- CWE - CWE-311 - Missing Encryption of Sensitive Data

assess_the_problem

Ask Yourself Whether

- The disk contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For aws_cdk.aws_ec2.Volume:

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';

new Volume(this, 'unencrypted-explicit', {
      availabilityZone: 'us-west-2a',
      size: Size.gibibytes(1),
      encrypted: false // Sensitive
    });
```

```
import { Size } from 'aws-cdk-lib';
import { Volume } from 'aws-cdk-lib/aws-ec2';

new Volume(this, 'unencrypted-implicit', {
      availabilityZone: 'eu-west-1a',
      size: Size.gibibytes(1),
    }); // Sensitive as encryption is disabled by default
```

root_cause

Amazon Elastic Block Store (EBS) is a block-storage service for Amazon Elastic Compute Cloud (EC2). EBS volumes can be encrypted, ensuring the security of both data-at-rest and data-in-transit between an instance and its attached EBS storage. In the case that adversaries gain physical access to the storage medium they are not able to access the data. Encryption can be enabled for specific volumes or for all new volumes and snapshots. Volumes created from snapshots inherit their encryption configuration. A volume created from an encrypted snapshot will also be encrypted by default.

Character classes in regular expressions should not contain only one character

Clave: javascript:S6397

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If a character class contains only one character, the effect is the same as just writing the character without a character class.

Thus, having only one character in a character class is usually a simple oversight that remained after removing other characters of the class.

Noncompliant code example

```
/a[b]c/
/[\^]/
```

Compliant solution

```
/abc/
/\^/
/a[*]c/ // Compliant, see Exceptions
```

Exceptions

This rule does not raise when the character inside the class is a metacharacter. This notation is sometimes used to avoid escaping (e.g., `[.]{3}` to match three dots).

Non-interactive elements shouldn't have event handlers

Clave: javascript:S6847

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

To fix this issue, remove the event handler from the non-interactive element and attach it to an interactive element instead. If the element is not interactive, it should not have an event handler.

Noncompliant code example

```
<li onClick={() => void 0} />
<div onClick={() => void 0} role="listitem" />
```

Compliant solution

```
<div onClick={() => void 0} role="button" />
<div onClick={() => void 0} role="presentation" />
<input type="text" onClick={() => void 0} /> // Interactive element does not require role.
<button onClick={() => void 0} className="foo" /> // button is interactive.
<div onClick={() => void 0} role="button" aria-hidden /> // This is hidden from the screenreader.
```

resources

Documentation

- WCAG - WAI-ARIA Authoring Practices Guide - Design Patterns and Widgets
- MDN - ARIA Techniques

root_cause

Attaching event handlers to non-interactive HTML elements can lead to significant accessibility issues. These elements, such as `<div>` and `<span>`, are not designed to interact with assistive technologies like screen readers, making it difficult for users with disabilities to navigate and interact with the website. Additionally, these elements may not be focusable or provide visual feedback when interacted with, resulting in a confusing and potentially frustrating user experience. Therefore, to maintain an accessible and user-friendly website, event handlers should be used exclusively with interactive elements.

The rule only considers the handlers `onClick`, `onMouseDown`, `onMouseUp`, `onKeyPress`, `onKeyDown`, and `onKeyUp`.

introduction

Non-interactive HTML elements, such as `<div>` and `<span>`, are not designed to have event handlers. When these elements are given event handlers, it can lead to accessibility issues.

Non-interactive DOM elements should not have an interactive handler

Clave: javascript:S6848

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Non-interactive DOM elements are HTML elements that are not designed to be interacted with by the user, for instance `<div>`, `<span>`, and `<footer>`, etc. They are typically used to structure content and do not have built-in interactivity or keyboard accessibility.

Interactive handlers, on the other hand, are event handlers that respond to user interactions. These include handlers like `onClick`, `onKeyDown`, `onMouseUp`, and more. When these handlers are added to an HTML element, they make the element respond to the specified user interaction.

When non-interactive elements have interactive handlers, it can lead to several problems:

- Unexpected behavior: Non-interactive elements are not designed to be interactive, so adding interactive handlers can cause unexpected behavior. For example, non-interactive elements do not naturally receive keyboard focus, so keyboard users might not be able to activate the handler.
- Confusing for assistive technologies: Assistive technologies might not announce non-interactive elements with interactive handlers correctly. This can make it difficult for users to understand how to interact with the content.
- Violation of HTML standards: Using interactive handlers on non-interactive elements can be seen as a misuse of HTML, as it goes against the intended use of these elements.

By enforcing that interactive handlers are only used on interactive elements, this rule helps create a more predictable and user-friendly experience for all users, and ensures that web content adheres to accessibility standards and best practices.

how_to_fix

The simplest and most recommended way is to replace the non-interactive elements with interactive ones. If for some reason you can’t replace the non-interactive element, you can add an interactive `role` attribute to it and manually manage its keyboard event handlers and focus. Common interactive roles include `button`, `link`, `checkbox`, `menuitem`, `menuitemcheckbox`, `menuitemradio`, `option`, `radio`, `searchbox`, `switch`, and `textbox`.

Noncompliant code example

```
<div onClick={() => {}} />; // Noncompliant
```

Compliant solution

```
<div onClick={() => {}} role="button" />;
```

resources

Documentation

- WCAG - Name, Role, Value
- MDN web docs - WAI-ARIA Roles

Heading elements should have accessible content

Clave: javascript:S6850

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Heading elements
- MDN web docs - aria-hidden
- WCAG - Headings and Labels

root_cause

Heading elements are represented by the tags `<h1>` through `<h6>`, with `<h1>` being the highest level and `<h6>` being the lowest. These elements are used to structure the content of the page and create a hierarchical outline that can be followed by users and search engines alike.

In the context of accessibility, heading elements play a crucial role. They provide a way for users, especially those using assistive technologies like screen readers, to navigate through the content of a webpage. By reading out the headings, screen readers can give users an overview of the content and allow them to jump to the section they’re interested in.

If a heading element is empty, it can lead to confusion as it doesn’t provide any information about the content that follows. This can make navigation difficult for users relying on screen readers, resulting in a poor user experience and making the website less accessible for people with visual impairments.

Therefore, to ensure your website is accessible to all users, it’s important to always include meaningful content in your heading elements.

how_to_fix

Do not leave empty your heading elements.

Noncompliant code example

```
function JavaScript101() {
  return (
    <>
```

```
      <h1>JavaScript Programming Guide</h1>
      <p>An introduction to JavaScript programming and its applications.</p>

      <h2>JavaScript Basics</h2>
      <p>Understanding the basic concepts in JavaScript programming.</p>

      <h3>Variables</h3>
      <p>Explanation of what variables are and how to declare them in JavaScript.</p>

      <h3 aria-hidden>Data Types</h3> // Noncompliant
      <p>Overview of the different data types in JavaScript.</p>

      <h3 /> // Noncompliant
      <p>Understanding how to declare and use functions in JavaScript.</p>
    </>
  );
}
```

Compliant solution

```
function JavaScript101() {
  return (
    <>
      <h1>JavaScript Programming Guide</h1>
      <p>An introduction to JavaScript programming and its applications.</p>

      <h2>JavaScript Basics</h2>
      <p>Understanding the basic concepts in JavaScript programming.</p>

      <h3>Variables</h3>
      <p>Explanation of what variables are and how to declare them in JavaScript.</p>

      <h3>Data Types</h3>
      <p>Overview of the different data types in JavaScript.</p>

      <h3>Functions</h3>
      <p>Understanding how to declare and use functions in JavaScript.</p>
    </>
  );
}
```

Images should have a non-redundant alternate description

Clave: javascript:S6851

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

`alt` attributes, also known as "alt tags" or "alt descriptions," are used to specify alternative text that is rendered when an image cannot be displayed. They are crucial for improving web accessibility, as they provide a text description of images for users who rely on screen readers.

Screen readers announce the presence of an `<img>` element and read its `alt` attribute aloud to describe the image. If the `alt` attribute includes words like "image", "picture", or "photo", it leads to redundancy as the screen reader would repeat "image". For instance, an `alt` attribute like "image of a sunrise" would be read as "Image, image of a sunrise", unnecessarily repeating "image".

Instead, the `alt` attribute should focus on describing the content of the image, not the fact that it is an image. This makes the browsing experience more efficient and enjoyable for users of screen readers, as they receive a concise and meaningful description of the image without unnecessary repetition.

resources

Documentation

- MDN web docs - img element
- MDN web docs - alt property
- WebAIM - Alternative Text

how_to_fix

To fix this issue, you should revise the `alt` attribute of your `<img>` elements to remove any instances of the words "image", "picture", or "photo". Instead, provide a concise and accurate description of the image content that adds value for users who cannot see the image.

Noncompliant code example

```
function MyImage() {
    return <img src="sunrise.jpg" alt="image of a sunrise" />; // Noncompliant: "Image, image of a sunrise"
}
```

Compliant solution

```
function MyImage() {
    return <img src="sunrise.jpg" alt="a sunrise over a mountain range" />;
}
```

Elements with an interactive role should support focus

Clave: javascript:S6852

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- W3C - Creating a logical tab order through links, form controls, and objects
- W3C - Fundamental Keyboard Navigation Conventions
- MDN - ARIA: button role - Accessibility concerns

root_cause

Lack of focusability can hinder navigation and interaction with the website, resulting in an exclusionary user experience and possible violation of accessibility guidelines.

introduction

Interactive elements being focusable is vital for website accessibility. It enables users, including those using assistive technologies, to interact effectively with the website. Without this, some users may be unable to access certain features, leading to a poor user experience and potential non-compliance with accessibility standards.

how_to_fix

Ensure that all interactive elements on your website can receive focus. This can be achieved by using standard HTML interactive elements, or by assigning a `tabindex` attribute of "0" to custom interactive components.

Noncompliant code example

```
<!-- Element with mouse/keyboard handler has no tabindex -->
<span onclick="submitForm();" role="button">Submit</span>

<!-- Anchor element without href is not focusable -->
<a onclick="showNextPage();" role="button">Next page</a>
```

Compliant solution

```
<!-- Element with mouse handler has tabIndex -->
<span onClick="doSomething();" tabIndex="0" role="button">Submit</span>

<!-- Focusable anchor with mouse handler -->
<a href="javascript:void(0);" onClick="doSomething();"> Next page </a>
```

Label elements should have a text label and an associated control

Clave: javascript:S6853

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

If you have a pair of control and `<label>` elements, make sure that the `<label>` wraps the control element. If you lack a control element, add one.

It is strongly recommended to avoid using generated `id`s since they must be deterministic.

Noncompliant code example

```
<input type="text" />
<label>Favorite food</label>
```

Compliant solution

```
<label>
  <input type="text" />
  Favorite food
</label>
```

Noncompliant code example

```
<label>Favorite food</label>
```

Compliant solution

```
<label>
  <MyCustomInput />
  Favorite food
</label>
```

introduction

A `<label>` element should wrap a control element or have an `<htmlFor>` attribute referencing a control and text content.

resources

Documentation

- MDN web docs - The Label element
- W3C - Info and Relationships
- W3C - Labels or Instructions

root_cause

When a label element lacks a text label or an associated control, it can lead to several issues:

1. **Poor Accessibility**: Screen readers rely on correctly associated labels to describe the function of the form control. If the label is not properly associated with a control, it can make the form difficult or impossible for visually impaired users to understand or interact with.
2. **Confusing User Interface**: Labels provide users with clear instructions about what information is required in a form control. Without a properly associated label, users might not understand what input is expected, leading to confusion and potential misuse of the form.
3. **Code Maintainability**: Properly structured and labeled code is easier to read, understand, and maintain. When labels are not correctly associated, it can make the code more difficult to navigate and debug, especially for new developers or those unfamiliar with the codebase.

Control elements are: * `<input>` * `<meter>` * `<output>` * `<progress>` * `<select>` * `<textarea>`

Exceptions

Custom components may contain control elements, therefore label elements containing custom elements do not raise issues.

Imports should not use absolute paths

Clave: javascript:S6859

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Replace the absolute path with one that is relative to your current file.

Noncompliant code example

```
import { foo } from '/home/project/api/bar.js';
```

Compliant solution

```
import { foo } from '../api/bar.js';
```

root_cause

In Node.js, it’s possible to import modules by specifying an absolute path, such as `/lib/foo/bar.js`. However, this approach can limit the portability of your code, as it becomes tied to your computer’s file system. This could potentially lead to problems when the code is distributed, for instance, via NPM packages. Therefore, it’s advisable to use relative paths or module names for importing modules to enhance the portability and compatibility of your code across different systems.

resources

Documentation

- MDN web docs - import
- Node.js docs - ECMAScript modules

Constructors should not contain asynchronous operations

Clave: javascript:S7059

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Constructors should not be asynchronous because they are meant to initialize an instance of a class synchronously. While you can technically run a promise in a constructor, the instance will be returned before the asynchronous operation completes, which can lead to potential issues if the rest of your code expects the object to be fully initialized. While returning the promise may seem the solution for this problem, it is not standard practice and can lead to unexpected behavior. Returning a promise (or, in general, any object from another class) from a constructor can be misleading and is considered a bad practice as this leads to unexpected results with inheritance and the `instanceof` operator.

```
class MyClass {
  constructor() {
    Promise.resolve().then(() => this.data = fetchData()); // Noncompliant, this.data will be undefined in the new instance
  }
}
```

Instead, you can create an asynchronous method that performs the necessary asynchronous operations after the object has been constructed

```
class MyClass {
  constructor() {
    this.data = null;
  }

  async initialize() {
    this.data = await fetchData();
  }
}

(async () => {
  const myObject = new MyClass();
  await myObject.initialize();
})();
```

Otherwise, you can use a static factory method that returns a promise resolving to an instance of the class after performing the asynchronous operations.

```
class MyClass {
  constructor(data) {
    this.data = data;
  }

  static async create() {
    const data = await fetchData();
    return new MyClass(data);
  }
}

(async () => {
  const myObject = await MyClass.create();
})();
```

Using an asynchronous factory function or an initialization method provides a more conventional, readable, and maintainable approach to handling asynchronous initialization in JavaScript.

resources

Documentation

- MDN web docs - new
- MDN web docs - Promise
- MDN web docs - instanceof
- MDN web docs - Inheritance and the prototype chain

Module should not import itself

Clave: javascript:S7060

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Modules
- MDN web docs - import
- Node.js docs - Node.js require

root_cause

When a module imports itself it has no effect. This means that the import statement does nothing useful and serves no purpose. This can happen during refactoring or when a developer mistakenly imports the module itself.

To fix the problem remove the self-import statement.

```
// file: foo.js
import foo from './foo'; // Noncompliant

const foo = require('./foo'); // Noncompliant

// file: index.js
import index from '.'; // Noncompliant

const index = require('.'); // Noncompliant
```

Non-empty statements should change control flow or have at least one side-effect

Clave: javascript:S905

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Identify statements that do not contribute to the functionality of the code and verify if they are intended to be part of the logic. If they are, there is a bug to be fixed. If they are not, then they are redundant and should be removed.

Noncompliant code example

```
function getResult() {
    let result = 42;
    if (shouldBeZero()) {
        result == 0; // Noncompliant: no side effect, was an assignment intended?
    }
    return result;
}

var msg = "Hello, "
  "World!"; // Noncompliant; have we forgotten '+' operator on previous line?
```

Compliant solution

```
function getResult() {
    let result = 42;
    if (shouldBeZero()) {
        result = 0; // Compliant
    }
    return result;
}

var msg = "Hello, " +
  "World!"; // Compliant
```

resources

Standards

- CWE - CWE-482 Comparing instead of Assigning

root_cause

When writing code, it is important to ensure that each statement serves a purpose and contributes to the overall functionality of the program. When they have no side effects or do not change the control flow, they can either indicate a programming error or be redundant:

1. The code does not behave as intended: The statements are expected to have an effect but they do not. This can be caused by mistyping, copy-and-paste errors, etc.
2. The statements are residual after a refactoring.

Exceptions

The rule does not raise an issue on statements containing only a semicolon (`;`).

introduction

Statements with no side effects and no change of control flow do not contribute to the functionality of the code and can indicate a programming error.

"continue" should not be used

Clave: javascript:S909

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

`continue` is an unstructured control flow statement. It makes code less testable, less readable and less maintainable. Structured control flow statements such as `if` should be used instead.

Noncompliant code example

```
for (i = 0; i < 10; i++) {
  if (i == 5) {
    continue;  /* Noncompliant */
  }
  alert("i = " + i);
}
```

Compliant solution

```
for (i = 0; i < 10; i++) {
  if (i != 5) {  /* Compliant */
    alert("i = " + i);
  }
}
```

Function calls should not pass extra arguments

Clave: javascript:S930

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - The `arguments` object
- MDN web docs - Rest parameters
- MDN web docs - Spread syntax (`...`)

root_cause

When you call a function in JavaScript and provide more arguments than the function expects, the extra arguments are simply ignored by the function.

```
function sum(a, b) {
  return a + b;
}

sum(1, 2, 3); // Noncompliant: The last argument is unexpected and will be ignored
```

Passing extra arguments in JavaScript is not inherently "bad," but it can lead to some potential issues or confusion if not handled correctly:

- The function signature is an essential part of its interface. Passing extra arguments can obscure the function’s intended use and make it less clear what the function actually requires.
- This can lead to unexpected behavior, as the function might not work as intended or produce incorrect results.
- Code that passes extra arguments can become harder to understand and maintain, especially when revisiting it at a later time.
- Other developers might find it challenging to comprehend the function’s purpose if extra arguments are scattered throughout the codebase.
- If you refactor the function later or rely on an external library that changes the expected number of arguments, your code with extra arguments could break unexpectedly.

While it’s possible to pass extra arguments, it’s essential to note that accessing those extra arguments directly inside the function is not straightforward. One common approach to handling extra arguments is to use the `arguments` object, which is an array-like object available within all function scopes.

```
function sum() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}

sum(1, 2, 3); // Compliant
```

However, it’s generally recommended to use the rest parameter syntax (`...args`) or utilize other techniques like the spread operator to deal with variable numbers of arguments in a more readable and maintainable way.

```
function sum(...args) {
  return args.reduce((a,b) => a + b, 0);
}

sum(1, 2, 3); // Compliant
```

Exceptions

No issue is reported when `arguments` is used in the body of the function being called.

```
function doSomething(a, b) {
  compute(arguments);
}

doSomething(1, 2, 3); // Compliant
```

Track comments matching a regular expression

Clave: javascript:S124

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

This rule template can be used to create rules which will be triggered when a comment matches a given regular expression.

For example, one can create a rule with the regular expression `.*REVIEW.*` to match all comment containing "REVIEW".

Sections of code should not be commented out

Clave: javascript:S125

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Commented-out code distracts the focus from the actual executed code. It creates a noise that increases maintenance code. And because it is never executed, it quickly becomes out of date and invalid.

Commented-out code should be deleted and can be retrieved from source control history if required.

"if ... else if" constructs should end with "else" clauses

Clave: javascript:S126

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

This rule applies whenever an `if` statement is followed by one or more `else if` statements; the final `else if` should be followed by an `else` statement.

The requirement for a final `else` statement is defensive programming.

The `else` statement should either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final `default` clause in a `switch` statement.

Noncompliant code example

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
}
```

Compliant solution

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
} else {
  throw "Unexpected value for x";
}
```

Switch cases should end with an unconditional "break" statement

Clave: javascript:S128

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

resources

- CWE - CWE-484 - Omitted Break Statement in Switch

root_cause

When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.

Noncompliant code example

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:  // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ?
    doSomething();
  default:
    doSomethingElse();
    break;
}
```

Compliant solution

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:
    doSomething();
    break;
  default:
    doSomethingElse();
    break;
}
```

Exceptions

This rule is relaxed in the following cases:

```
switch (myVariable) {
  case 0:                                // Empty case used to specify the same behavior for a group of cases.
  case 1:
    doSomething();
    break;
  case 2:                                // Use of return statement
    return;
  case 3:                               // Ends with comment when fall-through is intentional
    console.log("this case falls through")
    // fall through
  case 4:                                // Use of throw statement
    throw new IllegalStateException();
  case 5:                                // Use of continue statement
    continue;
  default:                               // For the last case, use of break statement is optional
    doSomethingElse();
}
```

"switch" statements should have "default" clauses

Clave: javascript:S131

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

The requirement for a final `default` clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken.

Noncompliant code example

```
switch (param) {  //missing default clause
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}

switch (param) {
  default: // default clause should be the last one
    error();
    break;
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

Compliant solution

```
switch (param) {
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
  default:
    error();
    break;
}
```

Exceptions

The rule ignores `switch` statements where the discriminant is a TypeScript union and there is a `case` branch for each constituent.

```
type Season = 'Spring' | 'Summer' | 'Fall' | 'Winter';
let season: Season;
switch (season) {
  case 'Spring':
    wakeUp();
    break;
  case 'Summer':
    getOut();
    break;
  case 'Fall':
    saveFood();
    break;
  case 'Winter':
    sleep();
    break;
}
```

The same applies for TypeScript enums.

```
enum Direction {
  Up,
  Down
}

let dir: Direction;
switch (dir) {
  case Direction.Up:
    getUp();
    break;
  case Direction.Down:
    getDown();
    break;
}
```

resources

- CWE - CWE-478 - Missing Default Case in Switch Statement

Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply

Clave: javascript:S134

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

- Guard clauses in programming - one of the approaches to reducing the depth of nesting

how_to_fix

The following example demonstrates the behavior of the rule with the default threshold of 3 levels of nesting and one of the potential ways to fix the code smell by introducing guard clauses:

Noncompliant code example

```
if (condition1) {                  // Compliant - depth = 1
  /* ... */
  if (condition2) {                // Compliant - depth = 2
    /* ... */
    for (let i = 0; i < 10; i++) {  // Compliant - depth = 3
      /* ... */
      if (condition4) {            // Noncompliant - depth = 4, which exceeds the limit
        if (condition5) {          // Depth = 5, exceeding the limit, but issues are only reported on depth = 4
          /* ... */
        }
        return;
      }
    }
  }
}
```

Compliant solution

```
if (!condition1) {
  return;
}
/* ... */
if (!condition2) {
  return;
}
for (let i = 0; i < 10; i++) {
  /* ... */
  if (condition4) {
    if (condition5) {
      /* ... */
    }
    return;
  }
}
```

root_cause

Nested control flow statements such as `if`, `for`, `while`, `switch`, and `try` are often key ingredients in creating what's known as "Spaghetti code". This code smell can make your program difficult to understand and maintain.

When numerous control structures are placed inside one another, the code becomes a tangled, complex web. This significantly reduces the code's readability and maintainability, and it also complicates the testing process.

Loops should not contain more than a single "break" or "continue" statement

Clave: javascript:S135

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The use of `break` and `continue` statements increases the complexity of the control flow and makes it harder to understand the program logic. In order to keep a good program structure, they should not be applied more than once per loop.

This rule reports an issue when there is more than one `break` or `continue` statement in a loop. The code should be refactored to increase readability if there is more than one.

Noncompliant code example

```
for (var i = 1; i <= 10; i++) {  // Noncompliant - 2 continue - one might be tempted to add some logic in between
  if (i % 2 == 0) {
    continue;
  }

  if (i % 3 == 0) {
    continue;
  }

  alert("i = " + i);
}
```

Functions should not have too many lines of code

Clave: javascript:S138

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A function that grows too large tends to aggregate too many responsibilities.

Such functions inevitably become harder to understand and therefore harder to maintain.

Above a specific threshold, it is strongly advised to refactor into smaller functions which focus on well-defined tasks.

Those smaller functions will not only be easier to understand, but also probably easier to test.

Exceptions

This rule ignores Immediately Invoked Function Expressions (IIFE), which are functions that are created and invoked without ever being assigned a name.

```
(function () { // Ignored by this rule

  function open() {  // Classic function declaration; not ignored
    // ...
  }

  function read() {
    // ...
  }

  function readlines() {
    // ...
  }
})();
```

This rule also ignores React Functional Components, which are JavaScript functions named with a capital letter and returning a React element (JSX syntax).

```
function Welcome() {
  const greeting = 'Hello, World!';

  // ...

  return (
    <div className="Welcome">
      <p>{greeting}</p>
    </div>
  );
}
```

Comments should not be located at the end of lines of code

Clave: javascript:S139

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule verifies that single-line comments are not located at the ends of lines of code. The main idea behind this rule is that in order to be really readable, trailing comments would have to be properly written and formatted (correct alignment, no interference with the visual structure of the code, not too long to be visible) but most often, automatic code formatters would not handle this correctly: the code would end up less readable. Comments are far better placed on the previous empty line of code, where they will always be visible and properly formatted.

Noncompliant code example

```
var a1 = b + c; // This is a trailing comment that can be very very long
```

Compliant solution

```
// This very long comment is better placed before the line of code
var a2 = b + c;
```

introduction

This rule is deprecated, and will eventually be removed.

Tests should not be skipped without providing a reason

Clave: javascript:S1607

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Jasmine Documentation - xit
- Jest Documentation - test.skip
- Mocha Documentation - Inclusive tests
- Node.js Documentation - Skipping tests

how_to_fix

A comment should be added before, on, or after the line of the unit test explaining why the test was disabled. Alternatively, if the test is no longer relevant, it should be removed entirely.

Noncompliant code example

```
describe('foo', function() {
  it.skip('should do something', function(done) { // Noncompliant
    done();
  });
});
```

Compliant solution

```
describe('foo', function() {
  // Reason: There is a bug in the code
  it.skip('should do something', function(done) { // Compliant
    done();
  });
});
```

how_to_fix

A comment should be added before, on, or after the line of the unit test explaining why the test was disabled. Alternatively, if the test is no longer relevant, it should be removed entirely.

Noncompliant code example

```
describe('foo', function() {
  test.skip('should do something', function(done) { // Noncompliant
    done();
  });
});
```

Compliant solution

```
describe('foo', function() {
  // Reason: There is a bug in the code
  test.skip('should do something', function(done) { // Compliant
    done();
  });
});
```

how_to_fix

A non-empty string literal should be passed to the skip options or as an argument to the call to skip (`{ skip: 'reason' }`) on the test context (`t.skip('reason')`), explaining why the test was disabled.

Noncompliant code example

```
const test = require('node:test');

test('should do something', { skip: true }, function(t) { // Noncompliant
  t.assert.ok(true);
});

test('should do something', function(t) {
  t.skip(); // Noncompliant
});
```

Compliant solution

```
const test = require('node:test');

test('should do something', { skip: 'There is a bug in the code' }, function(t) { // Compliant
  t.assert.ok(true);
});

test('should do something', function(t) {
  t.skip('There is a bug in the code'); // Compliant
});
```

how_to_fix

A comment should be added before, on, or after the line of the unit test explaining why the test was disabled. Alternatively, if the test is no longer relevant, it should be removed entirely.

Noncompliant code example

```
describe('foo', function() {
  xit('should do something', function(done) { // Noncompliant
    done();
  });
});
```

Compliant solution

```
describe('foo', function() {
  // Reason: There is a bug in the code
  xit('should do something', function(done) { // Compliant
    done();
  });
});
```

root_cause

Disabling unit tests lead to a lack of visibility into why tests are ignored, a decline in code quality as underlying problems remain unaddressed, and an increased maintenance burden due to the accumulation of disabled tests. It can also create a false sense of security about the stability of the codebase and pose challenges for new developers who may lack the context to understand why tests were disabled. Proper documentation and clear reasons for disabling tests are essential to ensure they are revisited and re-enabled once the issues are resolved.

This rule raises an issue when a test construct from Jasmine, Jest, Mocha, or Node.js Test Runner is disabled without providing an explanation. It relies on the presence of a package.json file and looks at the dependencies to determine which testing framework is used.

Objects should not be created to be dropped immediately without being used

Clave: javascript:S1848

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Creating an object without assigning it to a variable or using it in any function means the object is essentially created for no reason and may be dropped immediately without being used. Most of the time, this is due to a missing piece of code and could lead to an unexpected behavior.

If it’s intended because the constructor has side effects, that side effect should be moved into a separate method and called directly. This can help to improve the performance and readability of the code.

```
new MyConstructor(); // Noncompliant: object may be dropped
```

Determine if the objects are necessary for the code to function correctly. If they are not required, remove them from the code. Otherwise, assign them to a variable for later use.

```
let something = new MyConstructor();
```

Exceptions

- Creating new objects inside a `try` block is ignored.

```
try {
  new MyConstructor();
} catch (e) {
  /* ... */
}
```

- Known constructors with side effects like `Notification` or `Vue` are also ignored.

resources

Documentation

- MDN web docs - `Object.prototype.constructor`
- MDN web docs - constructor
- MDN web docs - `new` operator

Unused assignments should be removed

Clave: javascript:S1854

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Remove the unnecesarry assignment, then test the code to make sure that the right-hand side of a given assignment had no side effects (e.g. a method that writes certain data to a file and returns the number of written bytes).

Noncompliant code example

```
function foo(y) {
  let x = 100; // Noncompliant: dead store
  x = 150;     // Noncompliant: dead store
  x = 200;
  return x + y;
}
```

Compliant solution

```
function foo(y) {
  let x = 200; // Compliant: no unnecessary assignment
  return x + y;
}
```

root_cause

Dead stores refer to assignments made to local variables that are subsequently never used or immediately overwritten. Such assignments are unnecessary and don’t contribute to the functionality or clarity of the code. They may even negatively impact performance. Removing them enhances code cleanliness and readability. Even if the unnecessary operations do not do any harm in terms of the program’s correctness, they are - at best - a waste of computing resources.

Exceptions

The rule ignores

- Initializations to `-1`, `0`, `1`, `undefined`, `[]`, `{}`, `true`, `false` and `""`.
- Variables that start with an underscore (e.g. `_unused`) are ignored.
- Assignment of `null` is ignored because it is sometimes used to help garbage collection
- Increment and decrement expressions are ignored because they are often used idiomatically instead of `x+1`
- This rule also ignores variables declared with object destructuring using rest syntax (used to exclude some properties from object)

```
let {a, b, ...rest} = obj;  // 'a' and 'b' are compliant
doSomething(rest);

let [x1, x2, x3] = arr;     // 'x1' is noncompliant, as omitting syntax can be used: "let [, x2, x3] = arr;"
doSomething(x2, x3);
```

resources

Standards

- CWE - CWE-563 - Assignment to Variable without Use ('Unused Variable')

Related rules

- S1763 - All code should be reachable
- S2589 - Boolean expressions should not be gratuitous
- S3516 - Function returns should not be invariant
- S3626 - Jump statements should not be redundant

"if/else if" chains and "switch" cases should not have the same condition

Clave: javascript:S1862

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `if...else`
- MDN web docs - `switch`
- Wikipedia - Unreachable code

root_cause

Both `if-else` chains and `switch` statements are used for conditional branching, but they differ in their syntax and the way they handle multiple conditions.

- In an `if-else` chain, each condition is checked in order, and only the block associated with the first true condition is executed. If no condition is true, the code inside the `else` block (if present) will be executed.
- In a `switch` statement, the expression is evaluated once, and its value is compared against each case. If a matching case is found, the corresponding block of code is executed. The `break` statement is used to exit the `switch` block after a match. If no case matches the

expression, the code inside the `default` block (if present) will be executed.

Having the same condition in both `if-else` chains and `switch` cases can lead to unreachable code and a potential source of bugs. It defeats the purpose of conditional branching and can make the code harder to read and maintain.

```
if (event === 1) {
  openWindow();
} else if (event === 2) {
  closeWindow();
} else if (event === 1) {  // Noncompliant: Duplicated condition 'event === 1'
  moveWindowToTheBackground();
}

switch (event) {
  case 1:
    openWindow();
    break;
  case 2:
    closeWindow();
    break;
  case 1: // Noncompliant: Duplicated case '1'
    moveWindowToTheBackground();
    break;
}
```

Carefully review your conditions and ensure that they are not duplicated across the `if-else` chain or `switch` statement. Use distinct conditions and default blocks to cover all scenarios without redundant checks.

```
if (event === 1) {
  openWindow();
} else if (event === 2) {
  closeWindow();
} else if (event === 3) {
  moveWindowToTheBackground();
}

switch (event) {
  case 1:
    openWindow();
    break;
  case 2:
    closeWindow();
    break;
  case 3:
    moveWindowToTheBackground();
    break;
}
```

Variables should be declared explicitly

Clave: javascript:S2703

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Variable declaration is the process of creating a new variable and specifying its name. JavaScript provides three ways to declare variables: using the `var`, `let`, and `const` keywords.

- The `var` keyword is used to declare function-scoped or global-scoped variables, i.e. they are accessible throughout the function or the entire program, respectively.
- The `let` keyword is used to declare block-scoped variables, that is, variables accessible only within the nearest curly braces block where it is defined.
- The `const` keyword is used to declare variables that are constant, meaning their values cannot be reassigned.

Explicitly declaring variables improves code readability and maintainability. It makes it clear to other developers that you are creating a new variable and sets expectations about its scope. It also helps catch typos and avoid potential issues caused by accidentally reusing variable names.

If you assign a value to a variable without declaring it with `var`, `let`, or `const`, JavaScript treats it as an implicit global variable. Implicit globals can lead to unintended consequences and make it difficult to track and manage variables. They can cause naming conflicts, make code harder to understand, and introduce bugs that are hard to trace.

```
function f() {
  i = 1; // Noncompliant: i is global

  for (j = 0; j < array.length; j++) { // Noncompliant: j is global too
    // ...
  }
}
```

You should explicitly declare all the variables of your code. Use the `const` keyword if the variable is only assigned once and the `let` keyword otherwise.

```
function f() {
  const i = 1;

  for (let j = 0; j < array.length; j++) {
    // ...
  }
}
```

resources

Documentation

- MDN web docs - Variable
- MDN web docs - Declaring variables
- MDN web docs - Variable scope
- MDN web docs - var
- MDN web docs - let
- MDN web docs - const

Variables and functions should not be redeclared

Clave: javascript:S2814

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Redeclaration refers to the act of declaring a variable or function with the same name more than once within the same scope. In JavaScript, variable and function redeclarations are allowed but can lead to unexpected behavior and potential bugs in your code.

- Function declarations can be redeclared using the `function` keyword. In this case, the latest function declaration will overwrite the previous one.
- Variables declared using `var` can be redeclared within the same scope without any errors. The subsequent redeclaration will not affect the previous variable.
- Variable declarations with `var` in the same scope as a function named the same override the function's value.

This rule checks that a declaration doesn't use a name already in use, whether variables, functions, or parameters. Such redeclarations are misleading and could have been made by mistake, the developer not realizing that the new assignment overwrites the symbol value.

```
var a = 'foo';
function a() {} // Noncompliant: Overriden by the variable 'a'
console.log(a); // prints "foo"

function myFunc(arg) {
  var arg = "event"; // Noncompliant: Shadows the parameter 'arg'
}

fun(); // prints "bar"

function fun() {
  console.log("foo");
}

fun(); // prints "bar"

function fun() { // Noncompliant: Replaces the previous declaration of 'fun'
  console.log("bar");
}

fun(); // prints "bar"
```

To avoid issues with variable and function redeclarations, you should use unique names as much as possible and declare variables with `let` and `const` only.

```
let a = 'foo';
function otherName() {}
console.log(a);

function myFunc(arg) {
  const newName = "event";
}

fun(); // prints "foo"

function fun() {
  print("foo");
}

fun(); // prints "foo"
```

```
function printBar() {
  print("bar");
}

printBar(); // prints "bar"
```

resources

Documentation

- MDN - Variable redeclarations
- MDN - Function redeclarations
- MDN - `var`
- MDN - `let`
- MDN - `const`

Web SQL databases should not be used

Clave: javascript:S2817

Severidad: BLOCKER

Estado: ⚠️ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

resources

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A9 - Using Components with Known Vulnerabilities

root_cause

The Web SQL Database standard never saw the light of day. It was first formulated, then deprecated by the W3C and was only implemented in some browsers. (It is not supported in Firefox or IE.)

Further, the use of a Web SQL Database poses security concerns, since you only need its name to access such a database.

Noncompliant code example

```
var db = window.openDatabase("myDb", "1.0", "Personal secrets stored here", 2*1024*1024);  // Noncompliant
```

Origins should be verified during cross-origin communications

Clave: javascript:S2819

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - postMessage API

Standards

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- CWE - CWE-345 - Insufficient Verification of Data Authenticity

introduction

Cross-origin communication allows different websites to interact with each other. This interaction is typically achieved through mechanisms like AJAX requests, WebSockets, or postMessage API. However, a vulnerability can arise when these communications are not properly secured by verifying their origins.

root_cause

Without origin verification, the target website cannot distinguish between legitimate requests from its own pages and malicious requests from an attacker's site. The attacker can craft a malicious website or script that sends requests to a target website where the user is already authenticated.

This vulnerability class is not about a single specific user input or action, but rather a series of actions that lead to an insecure cross-origin communication.

What is the potential impact?

The absence of origin verification during cross-origin communications can lead to serious security issues.

Data Breach

If an attacker can successfully exploit this vulnerability, they may gain unauthorized access to sensitive data. For instance, a user's personal information, financial details, or other confidential data could be exposed. This not only compromises the user's privacy but can also lead to identity theft or financial loss.

Unauthorized Actions

An attacker could manipulate the communication between websites to perform actions on behalf of the user without their knowledge. This could range from making unauthorized purchases to changing user settings or even deleting accounts.

how_to_fix

When sending a message, avoid using * for the target origin (it means no preference). Instead define it explicitly so the message will only be dispatched to this URI. When receiving the message, verify the orgin to be sure that it is sent by an authorized sender.

Noncompliant code example

When sending a message:

```
var iframe = document.getElementById("testiframe");
iframe.contentWindow.postMessage("hello", "*"); // Noncompliant: * is used
```

When receiving a message:

```
window.addEventListener("message", function(event) { // Noncompliant: no checks are done on the origin property.
  console.log(event.data);
 });
```

Compliant solution

When sending a message:

```
var iframe = document.getElementById("testiframe");
iframe.contentWindow.postMessage("hello", "https://secure.example.com");
```

When receiving a message:

```
window.addEventListener("message", function(event) {
  if (event.origin !== "http://example.org")
    return;

  console.log(event.data)
});
```

Functions should always return the same type

Clave: javascript:S3800

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Unlike strongly typed languages, JavaScript does not enforce a return type on a function. This means that different paths through a function can return different types of values.

Returning different types from a function can make the code less readable and harder to understand. Maintainers may have to spend more time figuring out how the function works and what it returns. Additionally, it can be harder to ensure that the code is free of type-related errors.

```
function foo(a) {  // Noncompliant: function returns 'boolean' or 'number'
  if (a === 1) {
    return true;
  }
  return 3;
}
```

Rework the function so that it always returns the same type. This makes the code more consistent and easier to understand. Maintainers can rely on the function to behave in a predictable way.

```
function foo(a) {
  if (a === 1) {
    return true;
  }
  return false;
}
```

Exceptions

- Functions returning `this` are ignored.

```
function foo() {
  // ...
  return this;
}
```

- Functions returning expressions having type `any` are ignored.

resources

Documentation

- MDN web docs - `return`

Functions should use "return" consistently

Clave: javascript:S3801

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Unlike strongly typed languages, JavaScript does not enforce a return type on a function. This means that different paths through a function can return different types of values, which can be very confusing to the user and significantly harder to maintain.

In particular a function, in JavaScript, will return `undefined` in any of the following cases:

- It exits without a `return` statement.
- It executes a `return` with no value.

This rule verifies that return values are either always or never specified for each path through a function.

Noncompliant code example

```
function foo(a) { // Noncompliant, function exits without "return"
  if (a == 1) {
    return true;
  }
}
```

Compliant solution

```
function foo(a) {
  if (a == 1) {
    return true;
  }
  return false;
}
```

All branches in a conditional structure should not have exactly the same implementation

Clave: javascript:S3923

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Having all branches of a `switch` or `if` chain with the same implementation indicates a problem.

In the following code:

```
if (b == 0) {  // Noncompliant
  doOneMoreThing();
}
else {
  doOneMoreThing();
}

let a = (b == 0) ? getValue() : getValue();   // Noncompliant

switch (i) {  // Noncompliant
  case 1:
    doSomething();
    break;
  case 2:
    doSomething();
    break;
  case 3:
    doSomething();
    break;
  default:
    doSomething();
}
```

Either there is a copy-paste error that needs fixing or an unnecessary `switch` or `if` chain that should be removed.

Exceptions

This rule does not apply to `if` chains without `else`, nor to `switch` without a `default` clause.

```
if(b == 0) {    //no issue, this could have been done on purpose to make the code more readable
  doSomething();
} else if(b == 1) {
  doSomething();
}
```

"await" should only be used with promises

Clave: javascript:S4123

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - await
- MDN web docs - Promise
- MDN web docs - Conversion to promise

root_cause

Promises are objects that represent the eventual completion or failure of an asynchronous operation. They provide a way to handle asynchronous operations in a more organized and manageable manner. To use `await`, you need to ensure that you are calling a function or an expression that returns a promise.

The reason `await` should only be used on a promise is that it expects the operand to be a promise object. When you use `await`, it waits for the promise to be resolved or rejected and then returns the resolved value or throws the rejection reason, respectively. If the operand of `await` is not a promise, awaiting it is redundant and might not have been the developer’s intent.

If you try to use `await` on a non-promise value, such as a regular object or a primitive type, it will not pause the execution of the function because there is no asynchronous behavior involved. Instead, `await` will convert the value to a resolved promise, and waits for it.

```
const x = 42;
await x; // Noncompliant: x is a number, not a promise
```

You should only use `await` on promises because it is designed to handle asynchronous operations and works in conjunction with the Promise API to provide a clean and intuitive way to write asynchronous code in JavaScript.

```
const x = Promise.resolve(42);
await x;
```

When calling a function that returns a promise as the last expression, you might forget to return it, especially if you refactored your code from a single-expression arrow function.

```
function foo() {
  Promise.resolve(42);
}
async function bar() {
  await foo(); // Noncompliant
}
```

Make sure that you return the promise.

```
function foo() {
  return Promise.resolve(42);
}
async function bar() {
  await foo(); // Compliant
}
```

Exceptions

The rule does not raise issues if you are awaiting a function whose definition contains JSdoc with `@returns` or `@return` tags. This is due to JSdoc often mistakenly declaring a returning type without mentioning that it is resolved by a promise. For example:

```
async function foo () {
  await bar(); // Compliant
}

/**
 * @return {number}
 */
async function bar () {
  return 42;
}
```

Expanding archive files without controlling resource consumption is security-sensitive

Clave: javascript:S5042

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

- Define and control the ratio between compressed and uncompressed data, in general the data compression ratio for most of the legit archives is 1 to 3.
- Define and control the threshold for maximum total size of the uncompressed data.
- Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold, in particular it’s not recommended to recursively expand archives (an entry of an archive could be also an archive).

Compliant Solution

For tar module:

```
const tar = require('tar');
const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB

let fileCount = 0;
let totalSize = 0;

tar.x({
  file: 'foo.tar.gz',
  filter: (path, entry) => {
    fileCount++;
    if (fileCount > MAX_FILES) {
      throw 'Reached max. number of files';
    }

    totalSize += entry.size;
    if (totalSize > MAX_SIZE) {
      throw 'Reached max. size';
    }

    return true;
  }
});
```

For adm-zip module:

```
const AdmZip = require('adm-zip');
const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

let fileCount = 0;
let totalSize = 0;
let zip = new AdmZip("./foo.zip");
let zipEntries = zip.getEntries();
zipEntries.forEach(function(zipEntry) {
    fileCount++;
    if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
    }

    let entrySize = zipEntry.getData().length;
    totalSize += entrySize;
    if (totalSize > MAX_SIZE) {
        throw 'Reached max. size';
    }

    let compressionRatio = entrySize / zipEntry.header.compressedSize;
    if (compressionRatio > THRESHOLD_RATIO) {
        throw 'Reached max. compression ratio';
    }

    if (!zipEntry.isDirectory) {
        zip.extractEntryTo(zipEntry.entryName, ".");
    }
});
```

For jszip module:

```
const fs = require("fs");
const pathmodule = require("path");
const JSZip = require("jszip");

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB

let fileCount = 0;
let totalSize = 0;
let targetDirectory = __dirname + '/archive_tmp';

fs.readFile("foo.zip", function(err, data) {
  if (err) throw err;
  JSZip.loadAsync(data).then(function (zip) {
    zip.forEach(function (relativePath, zipEntry) {
      fileCount++;
      if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
      }

      // Prevent ZipSlip path traversal (S6096)
      const resolvedPath = pathmodule.join(targetDirectory, zipEntry.name);
      if (!resolvedPath.startsWith(targetDirectory)) {
        throw 'Path traversal detected';
      }

      if (!zip.file(zipEntry.name)) {
        fs.mkdirSync(resolvedPath);
      } else {
        zip.file(zipEntry.name).async('nodebuffer').then(function (content) {
          totalSize += content.length;
          if (totalSize > MAX_SIZE) {
            throw 'Reached max. size';
          }

          fs.writeFileSync(resolvedPath, content);
        });
      }
    });
  });
});
```

Be aware that due to the similar structure of sensitive and compliant code the issue will be raised in both cases. It is up to the developer to decide if the implementation is secure.

For yauzl module

```
const yauzl = require('yauzl');

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

yauzl.open('foo.zip', function (err, zipfile) {
  if (err) throw err;

  let fileCount = 0;
  let totalSize = 0;

  zipfile.on("entry", function(entry) {
    fileCount++;
    if (fileCount > MAX_FILES) {
      throw 'Reached max. number of files';
    }

    // The uncompressedSize comes from the zip headers, so it might not be trustworthy.
    // Alternatively, calculate the size from the readStream.
    let entrySize = entry.uncompressedSize;
    totalSize += entrySize;
    if (totalSize > MAX_SIZE) {
      throw 'Reached max. size';
    }

    if (entry.compressedSize > 0) {
      let compressionRatio = entrySize / entry.compressedSize;
      if (compressionRatio > THRESHOLD_RATIO) {
        throw 'Reached max. compression ratio';
      }
    }

    zipfile.openReadStream(entry, function(err, readStream) {
      if (err) throw err;
      // TODO: extract
    });
  });
});
```

Be aware that due to the similar structure of sensitive and compliant code the issue will be raised in both cases. It is up to the developer to decide if the implementation is secure.

For extract-zip module:

```
const extract = require('extract-zip')

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

async function main() {
  let fileCount = 0;
  let totalSize = 0;

  let target = __dirname + '/foo';
  await extract('foo.zip', {
    dir: target,
    onEntry: function(entry, zipfile) {
      fileCount++;
      if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
      }

      // The uncompressedSize comes from the zip headers, so it might not be trustworthy.
      // Alternatively, calculate the size from the readStream.
      let entrySize = entry.uncompressedSize;
      totalSize += entrySize;
      if (totalSize > MAX_SIZE) {
        throw 'Reached max. size';
      }

      if (entry.compressedSize > 0) {
        let compressionRatio = entrySize / entry.compressedSize;
        if (compressionRatio > THRESHOLD_RATIO) {
          throw 'Reached max. compression ratio';
        }
      }
    }
  });
}
main();
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification)
- bamsoftware.com - A better Zip Bomb

assess_the_problem

Ask Yourself Whether

Archives to expand are untrusted and:

- There is no validation of the number of entries in the archive.
- There is no validation of the total size of the uncompressed data.
- There is no validation of the ratio between the compressed and uncompressed archive entry.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For tar module:

```
const tar = require('tar');

tar.x({ // Sensitive
  file: 'foo.tar.gz'
});
```

For adm-zip module:

```
const AdmZip = require('adm-zip');

let zip = new AdmZip("./foo.zip");
zip.extractAllTo("."); // Sensitive
```

For jszip module:

```
const fs = require("fs");
const JSZip = require("jszip");

fs.readFile("foo.zip", function(err, data) {
  if (err) throw err;
  JSZip.loadAsync(data).then(function (zip) { // Sensitive
    zip.forEach(function (relativePath, zipEntry) {
      if (!zip.file(zipEntry.name)) {
        fs.mkdirSync(zipEntry.name);
      } else {
        zip.file(zipEntry.name).async('nodebuffer').then(function (content) {
          fs.writeFileSync(zipEntry.name, content);
        });
      }
    });
  });
});
```

For yauzl module

```
const yauzl = require('yauzl');

yauzl.open('foo.zip', function (err, zipfile) {
  if (err) throw err;

  zipfile.on("entry", function(entry) {
    zipfile.openReadStream(entry, function(err, readStream) {
      if (err) throw err;
      // TODO: extract
    });
  });
});
```

For extract-zip module:

```
const extract = require('extract-zip')

async function main() {
  let target = __dirname + '/test';
  await extract('test.zip', { dir: target }); // Sensitive
}
main();
```

default

Successful Zip Bomb attacks occur when an application expands untrusted archive files without controlling the size of the expanded data, which can lead to denial of service. A Zip bomb is usually a malicious archive file of a few kilobytes of compressed data but turned into gigabytes of uncompressed data. To achieve this extreme compression ratio, attackers will compress irrelevant data (eg: a long string of repeated bytes).

Ask Yourself Whether

Archives to expand are untrusted and:

- There is no validation of the number of entries in the archive.
- There is no validation of the total size of the uncompressed data.
- There is no validation of the ratio between the compressed and uncompressed archive entry.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Define and control the ratio between compressed and uncompressed data, in general the data compression ratio for most of the legit archives is 1 to 3.
- Define and control the threshold for maximum total size of the uncompressed data.
- Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold, in particular it’s not recommended to recursively expand archives (an entry of an archive could be also an archive).

Sensitive Code Example

For tar module:

```
const tar = require('tar');

tar.x({ // Sensitive
  file: 'foo.tar.gz'
});
```

For adm-zip module:

```
const AdmZip = require('adm-zip');

let zip = new AdmZip("./foo.zip");
zip.extractAllTo("."); // Sensitive
```

For jszip module:

```
const fs = require("fs");
const JSZip = require("jszip");

fs.readFile("foo.zip", function(err, data) {
  if (err) throw err;
  JSZip.loadAsync(data).then(function (zip) { // Sensitive
    zip.forEach(function (relativePath, zipEntry) {
      if (!zip.file(zipEntry.name)) {
        fs.mkdirSync(zipEntry.name);
      } else {
        zip.file(zipEntry.name).async('nodebuffer').then(function (content) {
          fs.writeFileSync(zipEntry.name, content);
        });
      }
    });
  });
});
```

For yauzl module

```
const yauzl = require('yauzl');

yauzl.open('foo.zip', function (err, zipfile) {
  if (err) throw err;

  zipfile.on("entry", function(entry) {
    zipfile.openReadStream(entry, function(err, readStream) {
      if (err) throw err;
      // TODO: extract
    });
  });
});
```

For extract-zip module:

```
const extract = require('extract-zip')

async function main() {
  let target = __dirname + '/test';
  await extract('test.zip', { dir: target }); // Sensitive
}
main();
```

Compliant Solution

For tar module:

```
const tar = require('tar');
const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB

let fileCount = 0;
let totalSize = 0;

tar.x({
  file: 'foo.tar.gz',
  filter: (path, entry) => {
    fileCount++;
    if (fileCount > MAX_FILES) {
      throw 'Reached max. number of files';
    }

    totalSize += entry.size;
    if (totalSize > MAX_SIZE) {
      throw 'Reached max. size';
    }

    return true;
  }
});
```

For adm-zip module:

```
const AdmZip = require('adm-zip');
const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

let fileCount = 0;
let totalSize = 0;
let zip = new AdmZip("./foo.zip");
let zipEntries = zip.getEntries();
zipEntries.forEach(function(zipEntry) {
    fileCount++;
    if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
    }

    let entrySize = zipEntry.getData().length;
    totalSize += entrySize;
    if (totalSize > MAX_SIZE) {
        throw 'Reached max. size';
    }

    let compressionRatio = entrySize / zipEntry.header.compressedSize;
    if (compressionRatio > THRESHOLD_RATIO) {
        throw 'Reached max. compression ratio';
    }

    if (!zipEntry.isDirectory) {
        zip.extractEntryTo(zipEntry.entryName, ".");
    }
});
```

For jszip module:

```
const fs = require("fs");
const pathmodule = require("path");
const JSZip = require("jszip");

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB

let fileCount = 0;
let totalSize = 0;
let targetDirectory = __dirname + '/archive_tmp';

fs.readFile("foo.zip", function(err, data) {
  if (err) throw err;
  JSZip.loadAsync(data).then(function (zip) {
    zip.forEach(function (relativePath, zipEntry) {
      fileCount++;
      if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
      }

      // Prevent ZipSlip path traversal (S6096)
      const resolvedPath = pathmodule.join(targetDirectory, zipEntry.name);
      if (!resolvedPath.startsWith(targetDirectory)) {
        throw 'Path traversal detected';
      }

      if (!zip.file(zipEntry.name)) {
        fs.mkdirSync(resolvedPath);
      } else {
        zip.file(zipEntry.name).async('nodebuffer').then(function (content) {
          totalSize += content.length;
          if (totalSize > MAX_SIZE) {
            throw 'Reached max. size';
          }

          fs.writeFileSync(resolvedPath, content);
        });
      }
    });
  });
});
```

Be aware that due to the similar structure of sensitive and compliant code the issue will be raised in both cases. It is up to the developer to decide if the implementation is secure.

For yauzl module

```
const yauzl = require('yauzl');

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

yauzl.open('foo.zip', function (err, zipfile) {
  if (err) throw err;

  let fileCount = 0;
  let totalSize = 0;

  zipfile.on("entry", function(entry) {
    fileCount++;
    if (fileCount > MAX_FILES) {
      throw 'Reached max. number of files';
    }

    // The uncompressedSize comes from the zip headers, so it might not be trustworthy.
    // Alternatively, calculate the size from the readStream.
    let entrySize = entry.uncompressedSize;
    totalSize += entrySize;
    if (totalSize > MAX_SIZE) {
      throw 'Reached max. size';
    }

    if (entry.compressedSize > 0) {
      let compressionRatio = entrySize / entry.compressedSize;
      if (compressionRatio > THRESHOLD_RATIO) {
        throw 'Reached max. compression ratio';
      }
    }

    zipfile.openReadStream(entry, function(err, readStream) {
      if (err) throw err;
      // TODO: extract
    });
  });
});
```

Be aware that due to the similar structure of sensitive and compliant code the issue will be raised in both cases. It is up to the developer to decide if the implementation is secure.

For extract-zip module:

```
const extract = require('extract-zip')

const MAX_FILES = 10000;
const MAX_SIZE = 1000000000; // 1 GB
const THRESHOLD_RATIO = 10;

async function main() {
  let fileCount = 0;
  let totalSize = 0;

  let target = __dirname + '/foo';
  await extract('foo.zip', {
    dir: target,
    onEntry: function(entry, zipfile) {
      fileCount++;
      if (fileCount > MAX_FILES) {
        throw 'Reached max. number of files';
      }

      // The uncompressedSize comes from the zip headers, so it might not be trustworthy.
      // Alternatively, calculate the size from the readStream.
      let entrySize = entry.uncompressedSize;
      totalSize += entrySize;
      if (totalSize > MAX_SIZE) {
        throw 'Reached max. size';
      }

      if (entry.compressedSize > 0) {
        let compressionRatio = entrySize / entry.compressedSize;
        if (compressionRatio > THRESHOLD_RATIO) {
          throw 'Reached max. compression ratio';
        }
      }
    }
  });
}
main();
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification)
- bamsoftware.com - A better Zip Bomb

root_cause

Successful Zip Bomb attacks occur when an application expands untrusted archive files without controlling the size of the expanded data, which can lead to denial of service. A Zip bomb is usually a malicious archive file of a few kilobytes of compressed data but turned into gigabytes of uncompressed data. To achieve this extreme compression ratio, attackers will compress irrelevant data (eg: a long string of repeated bytes).

Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string

Clave: javascript:S6019

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a reluctant (or lazy) quantifier is followed by a pattern that can match the empty string or directly by the end of the regex, it will always match zero times for `*?` or one time for `+?`. If a reluctant quantifier is followed directly by the end anchor (`$`), it behaves indistinguishably from a greedy quantifier while being less efficient.

This is likely a sign that the regex does not work as intended.

Noncompliant code example

```
str.split(/.*?x?/); // Noncompliant, this will behave just like "x?"
/^.*?$/.test(str); // Noncompliant, replace with ".*"
```

Compliant solution

```
str.split(/.*?x/);
/^.*$/.test(str);
```

Tests should not execute any code after "done()" is called

Clave: javascript:S6079

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Mocha Documentation - Asynchronous code

root_cause

The `done` callback of Mocha is used to signal the end of an asynchronous test. It is called when the test is complete, either successfully or with an error. It is important not to follow the `done` callback with code because the test may not have completed yet, and the code may execute before the test is finished. This can lead to unpredictable results and make it difficult to debug issues.

It is recommended to use the `done` callback only to signal the end of the test and handle any necessary cleanup or assertions before the callback.

Here's a bad example of using Mocha's `done` callback:

```
const expect = require("chai").expect;

describe('My test suite', function() {
  it('should do something asynchronously', function(done) {
    setTimeout(function() {
        expect(2 + 2).to.equal(4);
        expect('hello').to.have.lengthOf(5);
        done();

        console.log('Test has completed.'); // Noncompliant: Code after calling done, which may produce unexpected behavior
    }, 1000);
  });
});
```

Since the `console.log` statement is executed after calling `done()`, there is no guarantee that it will run after the test has fully completed. It may be correctly executed, but it might as well be assigned to a different test, no test, or even completely ignored.

To fix this, the `done` callback should be called after the `console.log` statement.

```
const expect = require("chai").expect;

describe('My test suite', function() {
  it('should do something asynchronously', function(done) {
    setTimeout(function() {
        expect(2 + 2).to.equal(4);
        expect('hello').to.have.lengthOf(5);

        console.log('Test has completed.');
        done();
    }, 1000);
  });
});
```

Disabling server-side encryption of S3 buckets is security-sensitive

Clave: javascript:S6245

Severidad: MINOR

Estado: ⚠ Depreciada

Descripción: No disponible

root_cause

This rule is deprecated, and will eventually be removed.

Server-side encryption (SSE) encrypts an object (not the metadata) as it is written to disk (where the S3 bucket resides) and decrypts it as it is read from disk. This doesn’t change the way the objects are accessed, as long as the user has the necessary permissions, objects are retrieved as if they were unencrypted. Thus, SSE only helps in the event of disk thefts, improper disposals of disks and other attacks on the AWS infrastructure itself.

There are three SSE options:

- Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3)
  - AWS manages encryption keys and the encryption itself (with AES-256) on its own.
- Server-Side Encryption with Customer Master Keys (CMKs) Stored in AWS Key Management Service (SSE-KMS)
  - AWS manages the encryption (AES-256) of objects and encryption keys provided by the AWS KMS service.
- Server-Side Encryption with Customer-Provided Keys (SSE-C)
  - AWS manages only the encryption (AES-256) of objects with encryption keys provided by the customer. AWS doesn’t store the customer’s encryption keys.

default

This rule is deprecated, and will eventually be removed.

Server-side encryption (SSE) encrypts an object (not the metadata) as it is written to disk (where the S3 bucket resides) and decrypts it as it is read from disk. This doesn’t change the way the objects are accessed, as long as the user has the necessary permissions, objects are retrieved as if they were unencrypted. Thus, SSE only helps in the event of disk thefts, improper disposals of disks and other attacks on the AWS infrastructure itself.

There are three SSE options:

- Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3)
  - AWS manages encryption keys and the encryption itself (with AES-256) on its own.
- Server-Side Encryption with Customer Master Keys (CMKs) Stored in AWS Key Management Service (SSE-KMS)
  - AWS manages the encryption (AES-256) of objects and encryption keys provided by the AWS KMS service.
- Server-Side Encryption with Customer-Provided Keys (SSE-C)
  - AWS manages only the encryption (AES-256) of objects with encryption keys provided by the customer. AWS doesn’t store the customer’s encryption keys.

Ask Yourself Whether

- The S3 bucket stores sensitive information.
- The infrastructure needs to comply to some regulations, like HIPAA or PCI DSS, and other standards.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to use SSE. Choosing the appropriate option depends on the level of control required for the management of encryption keys.

Sensitive Code Example

Server-side encryption is not used:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'default'
}); // Sensitive
```

Bucket encryption is disabled by default.

Compliant Solution

Server-side encryption with Amazon S3-Managed Keys is used:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    encryption: s3.BucketEncryption.KMS_MANAGED
});

# Alternatively with a KMS key managed by the user.

new s3.Bucket(this, 'id', {
    encryption: s3.BucketEncryption.KMS_MANAGED,
    encryptionKey: access_key
});
```

See

- AWS documentation - Protecting data using server-side encryption
- AWS CDK version 2 - BucketEncryption

assess_the_problem

Ask Yourself Whether

- The S3 bucket stores sensitive information.
- The infrastructure needs to comply to some regulations, like HIPAA or PCI DSS, and other standards.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Server-side encryption is not used:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'default'
}); // Sensitive
```

Bucket encryption is disabled by default.

how_to_fix

Recommended Secure Coding Practices

It’s recommended to use SSE. Choosing the appropriate option depends on the level of control required for the management of encryption keys.

Compliant Solution

Server-side encryption with Amazon S3-Managed Keys is used:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    encryption: s3.BucketEncryption.KMS_MANAGED
});

# Alternatively with a KMS key managed by the user.

new s3.Bucket(this, 'id', {
    encryption: s3.BucketEncryption.KMS_MANAGED,
    encryptionKey: access_key
});
```

See

- AWS documentation - Protecting data using server-side encryption
- AWS CDK version 2 - BucketEncryption

Authorizing HTTP communications with S3 buckets is security-sensitive

Clave: javascript:S6249

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

It’s recommended to enforce HTTPS only access by setting `enforceSSL` property to `true`

Compliant Solution

```
const s3 = require('aws-cdk-lib/aws-s3');

const bucket = new s3.Bucket(this, 'example', {
    bucketName: 'example',
    versioned: true,
    publicReadAccess: false,
    enforceSSL: true
});
```

See

- AWS documentation - Enforce encryption of data in transit
- AWS Foundational Security Best Practices controls - S3 buckets should require requests to use Secure Socket Layer
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information
- AWS CDK version 2 - Bucket

root_cause

By default, S3 buckets can be accessed through HTTP and HTTPs protocols.

As HTTP is a clear-text protocol, it lacks the encryption of transported data, as well as the capability to build an authenticated connection. It means that a malicious actor who is able to intercept traffic from the network can read, modify or corrupt the transported content.

default

By default, S3 buckets can be accessed through HTTP and HTTPs protocols.

As HTTP is a clear-text protocol, it lacks the encryption of transported data, as well as the capability to build an authenticated connection. It means that a malicious actor who is able to intercept traffic from the network can read, modify or corrupt the transported content.

Ask Yourself Whether

- The S3 bucket stores sensitive information.
- The infrastructure has to comply with AWS Foundational Security Best Practices standard.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to enforce HTTPS only access by setting `enforceSSL` property to `true`

Sensitive Code Example

S3 bucket objects access through TLS is not enforced by default:

```
const s3 = require('aws-cdk-lib/aws-s3');

const bucket = new s3.Bucket(this, 'example'); // Sensitive
```

Compliant Solution

```
const s3 = require('aws-cdk-lib/aws-s3');

const bucket = new s3.Bucket(this, 'example', {
    bucketName: 'example',
    versioned: true,
    publicReadAccess: false,
    enforceSSL: true
});
```

See

- AWS documentation - Enforce encryption of data in transit
- AWS Foundational Security Best Practices controls - S3 buckets should require requests to use Secure Socket Layer
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information
- AWS CDK version 2 - Bucket

assess_the_problem

Ask Yourself Whether

- The S3 bucket stores sensitive information.
- The infrastructure has to comply with AWS Foundational Security Best Practices standard.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

S3 bucket objects access through TLS is not enforced by default:

```
const s3 = require('aws-cdk-lib/aws-s3');

const bucket = new s3.Bucket(this, 'example'); // Sensitive
```

Disabling versioning of S3 buckets is security-sensitive

Clave: javascript:S6252

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

It’s recommended to enable S3 versioning and thus to have the possibility to retrieve and restore different versions of an object.

Compliant Solution

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    versioned: true
});
```

See

- AWS documentation - Using versioning in S3 buckets
- AWS CDK version 2 - Using versioning in S3 buckets

default

S3 buckets can be versioned. When the S3 bucket is unversioned it means that a new version of an object overwrites an existing one in the S3 bucket.

It can lead to unintentional or intentional information loss.

Ask Yourself Whether

- The bucket stores information that require high availability.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to enable S3 versioning and thus to have the possibility to retrieve and restore different versions of an object.

Sensitive Code Example

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    versioned: false // Sensitive
});
```

The default value of `versioned` is `false` so the absence of this parameter is also sensitive.

Compliant Solution

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    versioned: true
});
```

See

- AWS documentation - Using versioning in S3 buckets
- AWS CDK version 2 - Using versioning in S3 buckets

root_cause

S3 buckets can be versioned. When the S3 bucket is unversioned it means that a new version of an object overwrites an existing one in the S3 bucket.

It can lead to unintentional or intentional information loss.

assess_the_problem

Ask Yourself Whether

- The bucket stores information that require high availability.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    versioned: false // Sensitive
});
```

The default value of `versioned` is `false` so the absence of this parameter is also sensitive.

Allowing public ACLs or policies on a S3 bucket is security-sensitive

Clave: javascript:S6281

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

By default S3 buckets are private, it means that only the bucket owner can access it.

This access control can be relaxed with ACLs or policies.

To prevent permissive policies or ACLs to be set on a S3 bucket the following booleans settings can be enabled:

- `blockPublicAcls`: to block or not public ACLs to be set to the S3 bucket.
- `ignorePublicAcls`: to consider or not existing public ACLs set to the S3 bucket.
- `blockPublicPolicy`: to block or not public policies to be set to the S3 bucket.
- `restrictPublicBuckets`: to restrict or not the access to the S3 endpoints of public policies to the principals within the bucket owner account.

The other attribute `BlockPublicAccess.BLOCK_ACLS` only turns on `blockPublicAcls` and `ignorePublicAcls`. The public policies can still affect the S3 bucket.

However, all of those options can be enabled by setting the `blockPublicAccess` property of the S3 bucket to `BlockPublicAccess.BLOCK_ALL`.

assess_the_problem

Ask Yourself Whether

- The S3 bucket stores sensitive data.
- The S3 bucket is not used to store static resources of websites (images, css …).
- Many users have the permission to set ACL or policy to the S3 bucket.
- These settings are not already enforced to true at the account level.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

By default, when not set, the `blockPublicAccess` is fully deactivated (nothing is blocked):

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket'
}); // Sensitive
```

This `block_public_access` allows public ACL to be set:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: new s3.BlockPublicAccess({
        blockPublicAcls         : false, // Sensitive
        blockPublicPolicy       : true,
        ignorePublicAcls        : true,
        restrictPublicBuckets   : true
    })
});
```

The attribute `BLOCK_ACLS` only blocks and ignores public ACLs:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: s3.BlockPublicAccess.BLOCK_ACLS // Sensitive
});
```

how_to_fix

Recommended Secure Coding Practices

It’s recommended to configure:

- `blockPublicAcls` to `True` to block new attempts to set public ACLs.
- `ignorePublicAcls` to `True` to block existing public ACLs.
- `blockPublicPolicy` to `True` to block new attempts to set public policies.
- `restrictPublicBuckets` to `True` to restrict existing public policies.

Compliant Solution

This `blockPublicAccess` blocks public ACLs and policies, ignores existing public ACLs and restricts existing public policies:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL
});
```

A similar configuration to the one above can be obtained by setting all parameters of the `blockPublicAccess`

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: new s3.BlockPublicAccess({
        blockPublicAcls         : true,
        blockPublicPolicy       : true,
        ignorePublicAcls        : true,
        restrictPublicBuckets   : true
    })
});
```

See

- AWS Documentation - Blocking public access to your Amazon S3 storage
- CWE - CWE-284 - Improper Access Control
- STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.
- AWS CDK version 2 - BlockPublicAccess

default

By default S3 buckets are private, it means that only the bucket owner can access it.

This access control can be relaxed with ACLs or policies.

To prevent permissive policies or ACLs to be set on a S3 bucket the following booleans settings can be enabled:

- `blockPublicAcls`: to block or not public ACLs to be set to the S3 bucket.
- `ignorePublicAcls`: to consider or not existing public ACLs set to the S3 bucket.
- `blockPublicPolicy`: to block or not public policies to be set to the S3 bucket.
- `restrictPublicBuckets`: to restrict or not the access to the S3 endpoints of public policies to the principals within the bucket owner account.

The other attribute `BlockPublicAccess.BLOCK_ACLS` only turns on `blockPublicAcls` and `ignorePublicAcls`. The public policies can still affect the S3 bucket.

However, all of those options can be enabled by setting the `blockPublicAccess` property of the S3 bucket to `BlockPublicAccess.BLOCK_ALL`.

Ask Yourself Whether

- The S3 bucket stores sensitive data.
- The S3 bucket is not used to store static resources of websites (images, css …).
- Many users have the permission to set ACL or policy to the S3 bucket.
- These settings are not already enforced to true at the account level.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to configure:

- `blockPublicAcls` to `True` to block new attempts to set public ACLs.
- `ignorePublicAcls` to `True` to block existing public ACLs.
- `blockPublicPolicy` to `True` to block new attempts to set public policies.
- `restrictPublicBuckets` to `True` to restrict existing public policies.

Sensitive Code Example

By default, when not set, the `blockPublicAccess` is fully deactivated (nothing is blocked):

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket'
}); // Sensitive
```

This `block_public_access` allows public ACL to be set:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: new s3.BlockPublicAccess({
        blockPublicAcls         : false, // Sensitive
        blockPublicPolicy       : true,
        ignorePublicAcls        : true,
        restrictPublicBuckets   : true
    })
});
```

The attribute `BLOCK_ACLS` only blocks and ignores public ACLs:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: s3.BlockPublicAccess.BLOCK_ACLS // Sensitive
});
```

Compliant Solution

This `blockPublicAccess` blocks public ACLs and policies, ignores existing public ACLs and restricts existing public policies:

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL
});
```

A similar configuration to the one above can be obtained by setting all parameters of the `blockPublicAccess`

```
const s3 = require('aws-cdk-lib/aws-s3');

new s3.Bucket(this, 'id', {
    bucketName: 'bucket',
    blockPublicAccess: new s3.BlockPublicAccess({
        blockPublicAcls         : true,
        blockPublicPolicy       : true,
        ignorePublicAcls        : true,
        restrictPublicBuckets   : true
    })
});
```

See

- AWS Documentation - Blocking public access to your Amazon S3 storage
- CWE - CWE-284 - Improper Access Control
- STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.
- AWS CDK version 2 - BlockPublicAccess

Disabling Vue.js built-in escaping is security-sensitive

Clave: javascript:S6299

Severidad: BLOCKER

Estado: ⚠️ Depreciada

Descripción: No disponible

default

This rule is deprecated, and will eventually be removed.

Vue.js framework prevents XSS vulnerabilities by automatically escaping HTML contents with the use of native API browsers like `innerText` instead of `innerHtml`.

It’s still possible to explicity use `innerHtml` and similar APIs to render HTML. Accidentally rendering malicious HTML data will introduce an XSS vulnerability in the application and enable a wide range of serious attacks like accessing/modifying sensitive information or impersonating other users.

Ask Yourself Whether

The application needs to render HTML content which:

- could be user-controlled and not previously sanitized.
- is difficult to understand how it was constructed.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Avoid injecting HTML content with `v-html` directive unless the content can be considered 100% safe, instead try to rely as much as possible on built-in auto-escaping Vue.js features.
- Take care when using the `v-bind:href` directive to set URLs which can contain malicious Javascript (`javascript:onClick(...)`).
- Event directives like `:onmouseover` are also prone to Javascript injection and should not be used with unsafe values.

Sensitive Code Example

When using Vue.js templates, the `v-html` directive enables HTML rendering without any sanitization:

```
<div v-html="htmlContent"></div> <!-- Noncompliant -->
```

When using a rendering function, the `innerHTML` attribute enables HTML rendering without any sanitization:

```
Vue.component('element', {
  render: function (createElement) {
    return createElement(
      'div',
      {
        domProps: {
          innerHTML: this.htmlContent, // Noncompliant
        }
      }
    );
  },
});
```

When using JSX, the `domPropsInnerHTML` attribute enables HTML rendering without any sanitization:

```
<div domPropsInnerHTML={this.htmlContent}></div> <!-- Noncompliant -->
```

Compliant Solution

When using Vue.js templates, putting the content as a child node of the element is safe:

```
<div>{{ htmlContent }}</div>
```

When using a rendering function, using the `innerText` attribute or putting the content as a child node of the element is safe:

```
Vue.component('element', {
  render: function (createElement) {
    return createElement(
      'div',
      {
        domProps: {
          innerText: this.htmlContent,
        }
      },
      this.htmlContent // Child node
    );
  },
});
```

When using JSX, putting the content as a child node of the element is safe:

```
<div>{this.htmlContent}</div>
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Vue.js - Security - Injecting HTML

how_to_fix

Recommended Secure Coding Practices

- Avoid injecting HTML content with `v-html` directive unless the content can be considered 100% safe, instead try to rely as much as possible on built-in auto-escaping Vue.js features.
- Take care when using the `v-bind:href` directive to set URLs which can contain malicious Javascript (`javascript:onClick(...)`).
- Event directives like `:onmouseover` are also prone to Javascript injection and should not be used with unsafe values.

Compliant Solution

When using Vue.js templates, putting the content as a child node of the element is safe:

```
<div>{{ htmlContent }}</div>
```

When using a rendering function, using the `innerText` attribute or putting the content as a child node of the element is safe:

```
Vue.component('element', {
  render: function (createElement) {
    return createElement(
      'div',
      {
        domProps: {
          innerText: this.htmlContent,
        }
      },
      this.htmlContent // Child node
    );
  },
});
```

When using JSX, putting the content as a child node of the element is safe:

```
<div>{this.htmlContent}</div>
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Vue.js - Security - Injecting HTML

assess_the_problem

Ask Yourself Whether

The application needs to render HTML content which:

- could be user-controlled and not previously sanitized.
- is difficult to understand how it was constructed.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

When using Vue.js templates, the `v-html` directive enables HTML rendering without any sanitization:

```
<div v-html="htmlContent"></div> <!-- Noncompliant -->
```

When using a rendering function, the `innerHTML` attribute enables HTML rendering without any sanitization:

```
Vue.component('element', {
  render: function (createElement) {
    return createElement(
      'div',
      {
        domProps: {
          innerHTML: this.htmlContent, // Noncompliant
        }
      }
    );
  },
});
```

When using JSX, the `domPropsInnerHTML` attribute enables HTML rendering without any sanitization:

```
<div domPropsInnerHTML={this.htmlContent}></div> <!-- Noncompliant -->
```

root_cause

This rule is deprecated, and will eventually be removed.

Vue.js framework prevents XSS vulnerabilities by automatically escaping HTML contents with the use of native API browsers like `innerText` instead of `innerHtml`.

It's still possible to explicity use `innerHtml` and similar APIs to render HTML. Accidentally rendering malicious HTML data will introduce an XSS vulnerability in the application and enable a wide range of serious attacks like accessing/modifying sensitive information or impersonating other users.

Policies granting access to all resources of an account are security-sensitive

Clave: javascript:S6304

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

A policy that allows identities to access all resources in an AWS account may violate the principle of least privilege. Suppose an identity has permission to access all resources even though it only requires access to some non-sensitive ones. In this case, unauthorized access and disclosure of sensitive information will occur.

Exceptions

- Should not be raised on key policies (when AWS KMS actions are used.)
- Should not be raised on policies not using any resources (if and only if all actions in the policy never require resources.)

assess_the_problem

Ask Yourself Whether

The AWS account has more than one resource with different levels of sensitivity.

A risk exists if you answered yes to this question.

Sensitive Code Example

The wildcard `"*"` is specified as the resource for this `PolicyStatement`. This grants the update permission for all policies of the account:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [
        new iam.PolicyStatement({
            effect: iam.Effect.ALLOW,
            actions: ["iam:CreatePolicyVersion"],
            resources: ["*"] // Sensitive
        })
    ]
})
```

how_to_fix

Recommended Secure Coding Practices

It’s recommended to apply the least privilege principle, i.e., by only granting access to necessary resources. A good practice to achieve this is to organize or tag resources depending on the sensitivity level of data they store or process. Therefore, managing a secure access control is less prone to errors.

Compliant Solution

Restrict the update permission to the appropriate subset of policies:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [
        new iam.PolicyStatement({
            effect: iam.Effect.ALLOW,
            actions: ["iam:CreatePolicyVersion"],
            resources: ["arn:aws:iam:::policy/team1/*"]
        })
    ]
})
```

See

- AWS Documentation - Grant least privilege
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control

default

A policy that allows identities to access all resources in an AWS account may violate the principle of least privilege. Suppose an identity has permission to access all resources even though it only requires access to some non-sensitive ones. In this case, unauthorized access and disclosure of sensitive information will occur.

Ask Yourself Whether

The AWS account has more than one resource with different levels of sensitivity.

A risk exists if you answered yes to this question.

Recommended Secure Coding Practices

It’s recommended to apply the least privilege principle, i.e., by only granting access to necessary resources. A good practice to achieve this is to organize or tag resources depending on the sensitivity level of data they store or process. Therefore, managing a secure access control is less prone to errors.

Sensitive Code Example

The wildcard `"*"` is specified as the resource for this `PolicyStatement`. This grants the update permission for all policies of the account:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [
        new iam.PolicyStatement({
            effect: iam.Effect.ALLOW,
            actions: ["iam:CreatePolicyVersion"],
            resources: ["*"] // Sensitive
        })
    ]
})
```

Compliant Solution

Restrict the update permission to the appropriate subset of policies:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [
        new iam.PolicyStatement({
            effect: iam.Effect.ALLOW,
            actions: ["iam:CreatePolicyVersion"],
            resources: ["arn:aws:iam:::policy/team1/*"]
        })
    ]
})
```

Exceptions

- Should not be raised on key policies (when AWS KMS actions are used.)
- Should not be raised on policies not using any resources (if and only if all actions in the policy never require resources.)

See

- AWS Documentation - Grant least privilege
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control

Exclusive tests should not be commited to version control

Clave: javascript:S6426

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When using testing frameworks like Mocha and Jest, appending `.only()` to the test function allows running a single test case for a file. Using `.only()` means no other test from this file is executed. This is useful when debugging a specific use case.

```
describe("MyClass", function () {
    it.only("should run correctly", function () { // Noncompliant
        /*...*/
    });
});
```

However, it should not be used in production or development, as it is likely a leftover from debugging and serves no purpose in those contexts. It is strongly recommended not to include `.only()` usages in version control.

```
describe("MyClass", function () {
    it("should run correctly", function () {
        /*...*/
    });
});
```

resources

Documentation

- Mocha Documentation - Exclusive tests
- Jest Documentation - `test.only()`
- Jest Documentation - `describe.only()`

Disallow `.bind()` and arrow functions in JSX props

Clave: javascript:S6480

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using `Function.prototype.bind` and arrows functions as attributes will negatively impact performance in React. Each time the parent is rendered, the function will be re-created and trigger a render of the component causing excessive renders and more memory use. Wrapping the function in a `useCallback` hook will avoid additional renders. This rule ignores `Refs`. This rule does not raise findings on DOM nodes since that may require wrapping the DOM in a component. Still, better performance can be achieved if this rule is respected in DOM nodes too.

Noncompliant code example

```
<Component onClick={this._handleClick.bind(this)}></Component>
```

```
<Component onClick={() => handleClick()}></Component>
```

Compliant solution

```
function handleClick() {
    //...
}
```

```
<Component onClick={handleClick}></Component>
```

Situation can become more complicated when you need to pass additional parameters to the handler. Consider following component printing the list of letters. Consider following non-compliant example

```
class Alphabet extends React.Component {
    handleClick(letter) {
        console.log(`clicked ${letter}`);
    }
    render() {
        return (<div><ul>
            {letters.map(letter =>
                <li key={letter} onClick={() => this.handleClick(letter)}>{letter}</li>
            )}
        </ul></div>)
    }
}
```

To avoid creating the arrow function you can factor out `li` element as separate child component and use `props` to pass the `letter` and `onClick` handler.

```
class Alphabet extends React.Component {
    handleClick(letter) {
        console.log(`clicked ${letter}`);
    }
    render() {
        return (<div><ul>
            {letters.map(letter =>
                <Letter key={letter} letter={letter} handleClick={this.handleClick}></Letter>
            )}
        </ul></div>)
    }
}

class Letter extends React.Component {
    constructor(props) {
        super(props);
        this.handleClick = this.handleClick.bind(this)
    }
    handleClick() {
        this.props.handleClick(this.props.letter);
    }
    render() {
        return <li onClick={this.handleClick}> {this.props.letter} </li>
    }
}
```

alternatively you could rewrite `Letter` as a function and use `useCallback`

```
function Letter({ handleClick, letter }) {
    const onClick = React.useCallback(() => handleClick(letter), [letter])

    return <li onClick={onClick}>{letter}</li>
}
```

resources

- Passing Functions to Components - React documentation

React Context Provider values should have stable identities

Clave: javascript:S6481

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Context Caveats
- React Documentation - Passing Data Deeply with Context
- React Documentation - `useCallback` hook
- React Documentation - `useMemo` hook

root_cause

Whenever the `value` property of React context changes, React will rerender the context and all its child nodes and consumers. In JavaScript, things like object literals or function expressions will create a new identity every time they are evaluated. Such constructions should not be directly used as context `value` because React will always consider they have changed. This can significantly impact performance.

```
function Component() {
  return (
    <SomeContext.Provider value={{foo: 'bar'}}> { /* Noncompliant: value is an object literal */ }
      <SomeComponent />
    </SomeContext.Provider>
  );
}
```

To avoid additional rerenders wrap the value in a `useMemo` hook. Use the `useCallback()` hook if the value is a function.

```
function Component() {
  const obj = useMemo(() => ({foo: 'bar'}), []); // value is cached by useMemo
  return (
    <SomeContext.Provider value={obj}> { /* Compliant */ }
      <SomeComponent />
    </SomeContext.Provider>
  );
}
```

JSX list components keys should match up between renders

Clave: javascript:S6486

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Rendering lists
- React Documentation - Recursing On Children
- MDN web docs - Crypto: randomUUID() method
- Wikipedia - UUID

Related rules

- S6477 - JSX list components should have a key property
- S6479 - JSX list components should not use array indexes as key

root_cause

To optimize the rendering of React list components, a unique identifier (UID) is required in the `key` attribute for each list item. This UID lets React identify the item throughout its lifetime. Using generated values like `Math.random()` or `Date.now()` is discouraged as their return value will differ between calls, causing the keys to not match up between renders, recreating the DOM. Also, this may cause bugs if values collide.

```
function Blog(props) {
  return (
    <ul>
      {props.posts.map((post) =>
        <li key={Math.random()}> <!-- Noncompliant: Since the 'key' will be different on each render, React will update the DOM unnecessarily -->
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

To fix it, use a string or a number that uniquely identifies the list item. The key must be unique among its siblings, not globally.

If the data comes from a database, database IDs are already unique and are the best option. Otherwise, use a counter or a UUID generator.

Avoid using array indexes since, even if they are unique, the order of the elements may change.

```
function Blog(props) {
  return (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

React's "findDOMNode" should not be used

Clave: javascript:S6788

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In React, `findDOMNode` is used to get the browser DOM node given a component instance. However, using `findDOMNode` is fragile because the connection between the JSX node and the code manipulating the corresponding DOM node is not explicit. For example, changing the external structure of returned JSX will affect the return value of `findDOMNode`. There are also other caveats when using `findDOMNode`.

```
import { Component } from 'react';
import { findDOMNode } from 'react-dom';

class AutoselectingInput extends Component {
  componentDidMount() {
    const input = findDOMNode(this); // Noncompliant: findDOMNode is deprecated
    input.select();
  }

  render() {
    return <input defaultValue="Hello" />
  }
}
```

Instead, one should get the component’s own DOM node from a ref. Pass your ref as the `ref` attribute to the JSX tag for which you want to get the DOM node. This tells React to put this `<input>`’s DOM node into `inputRef.current`.

Use `createRef` to manage a specific DOM node. In modern React without class components, the equivalent code would call `useRef` instead.

```
import { createRef, Component } from 'react';

class AutoselectingInput extends Component {
  inputRef = createRef(null);

  componentDidMount() {
    const input = this.inputRef.current; // Always points to the input element
    input.select();
  }

  render() {
    return (
      <input ref={this.inputRef} defaultValue="Hello" />
    );
  }
}
```

resources

Documentation

- React Documentation - `findDOMNode`
- React Documentation - `createRef`
- React Documentation - `useRef`
- React Documentation - Manipulating the DOM with Refs

React's "isMounted" should not be used

Clave: javascript:S6789

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

React `isMounted()` is primarily used to avoid calling `setState()` after a component has unmounted, because calling `setState()` after a component has unmounted will emit a warning. Checking `isMounted()` before calling `setState()` does eliminate the warning, but it also defeats the purpose of the warning, which is raising awareness that the app is still holding a reference to the component after the component has been unmounted.

When using ES6 classes, using `isMounted()` is already prohibited.

```
class MyComponent extends React.Component {
  componentDidMount() {
    mydatastore.subscribe(this);
  }
  dataHandler() {
    if (this.isMounted()) { // Noncompliant: isMounted() hides the error
      //...
    }
  }
  render() {
    //... calls dataHandler()
  }
};
```

Find places where `setState()` might be called after a component has unmounted, and fix them. Such situations most commonly occur due to callbacks, when a component is waiting for some data and gets unmounted before the data arrives. Ideally, any callbacks should be canceled in `componentWillUnmount`, before the component unmounts.

```
class MyComponent extends React.Component {
  componentDidMount() {
    mydatastore.subscribe(this);
  }
  dataHandler() {
    //...
  }
  render() {
    //...
  }
  componentWillUnmount() {
    mydatastore.unsubscribe(this);
  }
}
```

resources

Documentation

- React Documentation - `isMounted` is an Antipattern

Lines should not be too long

Clave: javascript:S103

Severidad: MAJOR

Estado: ⚠️ Depreciada

Descripción: No disponible

root_cause

Scrolling horizontally to see a full line of code lowers the code readability.

introduction

This rule is deprecated, and will eventually be removed.

Files should not have too many lines of code

Clave: javascript:S104

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a source file grows too much, it can accumulate numerous responsibilities and become challenging to understand and maintain.

Above a specific threshold, refactor the file into smaller files whose code focuses on well-defined tasks. Those smaller files will be easier to understand and test.

Tabulation characters should not be used

Clave: javascript:S105

Severidad: MINOR

Estado: ⚠️ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

The tab width can differ from one development environment to another. Using tabs may require other developers to configure their environment (text editor, preferences, etc.) to read source code.

That is why using spaces is preferable.

Standard outputs should not be used directly to log anything

Clave: javascript:S106

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure

root_cause

In software development, logs serve as a record of events within an application, providing crucial insights for debugging. When logging, it is essential to ensure that the logs are:

- easily accessible
- uniformly formatted for readability
- properly recorded
- securely logged when dealing with sensitive data

Those requirements are not met if a program directly writes to the standard outputs (e.g., console). That is why defining and using a dedicated logger is highly recommended.

The following noncompliant code:

```
function doSomething() {
  // ...
  console.log("My Message");
  // ...
}
```

In `Node.js` could be replaced by the `winston` logging library:

```
const winston = require("winston");

const logger = winston.createLogger({
  level: "debug",
  format: winston.format.json(),
  transports: [new winston.transports.Console()],
});


function doSomething() {
  // ...
  logger.info("My Message");
  // ...
}
```

Functions should not have too many parameters

Clave: javascript:S107

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Functions with a long parameter list are difficult to use because maintainers must figure out the role of each parameter and keep track of their position.

```
function setCoordinates(x1, y1, z1, x2, y2, z2) { // Noncompliant
    // ...
}
```

The solution can be to:

- Split the function into smaller ones

```
// Each function does a part of what the original setCoordinates function was doing, so confusion risks are lower
function setOrigin(x, y, z) {
   // ...
}

function setSize(width, height, depth) {
   // ...
}
```

- Find a better data structure for the parameters that group data in a way that makes sense for the specific application domain

```
// In geometry, Point is a logical structure to group data
let point1 = { x: x1, y: y1, z: z1};
let point2 = { x: x1, y: y1, z: z1};
setCoordinates(point1, point2);

function setCoordinates(p1, p2) {
   // ...
}
```

This rule raises an issue when a function has more parameters than the provided threshold.

Exceptions

The rule ignores TypeScript parameter properties when counting parameters:

```
class C {
  constructor(
    private param1: number,     // ignored
    param2: boolean,            // counted
    public param3: string,      // ignored
    readonly param4: string[],  // ignored
    param5: number | string     // counted
  ) {} // Compliant by exception
}
```

The rule also ignores Angular component constructors:

```
import { Component } from '@angular/core';

@Component({/* ... */})
class Component {
  constructor(p1, p2, p3, p4, p5) {} // Compliant by exception
}
```

Image, area, button with image and object elements should have an alternative text

Clave: javascript:S1077

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Add an alternative text to the HTML element.

Noncompliant code example

```
<img src="foo.png" /> <!-- missing `alt` attribute -->
<input type="image" src="bar.png" /> <!-- missing alternative text attribute -->
<input type="image" src="bar.png" alt="" /> <!-- empty alternative text attribute on <input> -->

<img src="house.gif" usemap="#map1"
    alt="rooms of the house." />
<map id="map1" name="map1">
  <area shape="rect" coords="0,0,42,42"
    href="bedroom.html"/> <!-- missing alternative text attribute on <area> -->
  <area shape="rect" coords="0,0,21,21"
    href="lounge.html" alt=""/> <!-- empty `alt` attribute on <area> -->
</map>

<object {...props} />  <!-- missing alternative text attribute on <area> -->
```

Compliant solution

```
<img src="foo.png" alt="Some textual description of foo.png" />
<input type="image" src="bar.png" aria-labelledby="Textual description of bar.png" />

<img src="house.gif" usemap="#map1"
    alt="rooms of the house." />
<map id="map1" name="map1">
  <area shape="rect" coords="0,0,42,42"
    href="bedroom.html" alt="Bedroom" />
  <area shape="rect" coords="0,0,21,21"
    href="lounge.html" aria-label="Lounge"/>
</map>

<object>My welcoming Bar</object>
```

resources

Documentation

- W3C - W3C WAI Web Accessibility Tutorials
- W3C - Providing text alternatives for the area elements of image maps
- W3C - Using alt attributes on images used as submit buttons
- W3C - Using alt attributes on img elements
- W3C - Using null alt text and no title attribute on img elements for images that AT should ignore
- W3C - Combining adjacent image and text links for the same resource
- W3C - Non-text Content
- W3C - Link Purpose (In Context)
- W3C - Link Purpose (Link Only)

root_cause

The `alt`, `aria-label` and `aria-labelledby` attributes provide a textual alternative to an image.

It is used whenever the actual image cannot be rendered.

Common reasons for that include:

- The image can no longer be found
- Visually impaired users using a screen reader software
- Image loading is disabled, to reduce data consumption on mobile phones

It is also very important not to set an alternative text attribute to a non-informative value. For example, `<img ... alt="logo">` is useless as it doesn't give any information to the user. In this case, as for any other decorative image, it is better to use a CSS background image instead of an `<img>` tag. If using CSS `background-image` is not possible, an empty `alt=""` is tolerated. See Exceptions below.

This rule raises an issue when:

- An `<img>` element has no `alt` attribute.
- An `<input type="image">` element has no `alt`, `aria-label` or `aria-labelledby` attribute or they hold an empty string.
- An `<area>` element within an image map has no `alt`, `aria-label` or `aria-labelledby` attribute.
- An `<object>` element has no inner text, `title`, `aria-label` or `aria-labelledby` attribute.

Exceptions

`<img>` elements with an empty string `alt=""` attribute won't raise any issue. However, this way should be used in two cases only:

When the image is decorative and it is not possible to use a CSS background image. For example, when the decorative `<img>` is generated via javascript with a source image coming from a database, it is better to use an `<img alt="">` tag rather than generate CSS code.

```
<li *ngFor="let image of images">
    <img [src]="image" alt="">
</li>
```

When the image is not decorative but its `alt` text would repeat a nearby text. For example, images contained in links should not duplicate the link's text in their `alt` attribute, as it would make the screen reader repeat the text twice.

```
<a href="flowers.html">
    <img src="tulip.gif" alt="" />
    A blooming tulip
</a>
```

In all other cases you should use CSS background images.

Nested blocks of code should not be left empty

Clave: javascript:S108

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

An empty code block is confusing. It will require some effort from maintainers to determine if it is intentional or indicates the implementation is incomplete.

```
for (let i = 0; i < length; i++) {}  // Noncompliant: is the block empty on purpose, or is code missing?
```

Removing or filling the empty code blocks takes away ambiguity and generally results in a more straightforward and less surprising code.

Exceptions

The rule ignores:

- code blocks that contain comments
- `catch` blocks

Magic numbers should not be used

Clave: javascript:S109

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Magic numbers make the code more complex to understand as it requires the reader to have knowledge about the global context to understand the number itself. Their usage may seem obvious when writing the code, but it may not be the case for another developer or later once the context faded away. -1, 0, and 1 are not considered magic numbers.

Exceptions

- the following numbers used in arithmetic operations: -1, 0, 1, as well as powers of 2 and 10
- time-related constants such as 24 and 60 are excluded
- numbers used in JSX elements are excluded
- enum values, default values, and other assignments are excluded
- arguments to `parseInt()` and `JSON.stringify()` are excluded
- numbers used in bitwise operations are excluded

how_to_fix

Replacing them with a constant allows us to provide a meaningful name associated with the value. Instead of adding complexity to the code, it brings clarity and helps to understand the context and the global meaning.

Noncompliant code example

```
function doSomething() {
  for (let i = 0; i < 4; i++) { // Noncompliant, 4 is a magic number
    // ...
  }
}
```

Compliant solution

```
function doSomething() {
  const numberOfCycles = 4;
  for (let i = 0; i < numberOfCycles; i++) { // Compliant
    // ...
  }
}
```

introduction

A magic number is a hard-coded numerical value that may lack context or meaning. They should not be used because they can make the code less readable and maintainable.

---

iFrames must have a title

Clave: javascript:S1090

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - iframe element
- WCAG - Bypass Blocks
- WCAG - Name, Role, Value

how_to_fix

To fix missing iframe titles, you simply need to add a `title` attribute to your `<iframe>` element. The value of this attribute should be a brief description of the iframe’s content.

Noncompliant code example

```
function iframe() {
    return (
        <iframe src="https://openweathermap.org"></iframe> // Noncompliant
    );
}
```

Compliant solution

```
function iframe() {
    return (
        <iframe src="https://openweathermap.org" title="Weather forecasts, nowcasts and history"></iframe>
    );
}
```

root_cause

An iframe, or inline frame, is an HTML document embedded inside another HTML document on a website. The iframe HTML element is often used to insert content from another source, such as an advertisement, into a web page.

In the context of web accessibility, `<iframe>`'s should have a `title` attribute. This is because screen readers for the visually impaired use this title to help users understand the content of the iframe.

Without a title, it can be difficult for these users to understand the context or purpose of the iframe’s content.

Files should end with a newline

Clave: javascript:S113

Severidad: MINOR

Estado: ⚠️ Depreciada

Descripción: No disponible

root_cause

Some tools work better when files end with a newline.

This rule simply generates an issue if it is missing.

For example, a Git diff looks like this if the empty line is missing at the end of the file:

```
+class Test {
+}
\ No newline at end of file
```

introduction

This rule is deprecated, and will eventually be removed.

Variable, property and parameter names should comply with a naming convention

Clave: javascript:S117

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

introduction

Variables, properties, and parameters should be named consistently to communicate intent and improve maintainability. Rename your variable, property, or parameter to follow your project’s naming convention to address this issue.

resources

Documentation

- MDN web docs - Guidelines for writing JavaScript code: Variable names
- Wikipedia - Naming Convention (programming)

Related rules

- S100 - Function and method names should comply with a naming convention
- S101 - Class names should comply with a naming convention

root_cause

A naming convention in software development is a set of guidelines for naming code elements like variables, functions, and classes.
Variables, properties, and parameters hold the meaning of the written code. Their names should be meaningful and follow a consistent and easily recognizable pattern. Adhering to a consistent naming convention helps to make the code more readable and understandable, which makes it easier to maintain and debug. It also ensures consistency in the code, especially when multiple developers are working on the same project.

This rule checks that variable, property, and parameter names match a provided regular expression.

What is the potential impact?

Inconsistent naming of variables, properties, and parameters can lead to several issues in your code:

- **Reduced Readability**: Inconsistent variable, property, and parameter names make the code harder to read and understand; consequently, it is more difficult to identify the purpose of each variable, spot errors, or comprehend the logic.
- **Difficulty in Identifying Variables**: The variables, properties, and parameters that don’t adhere to a standard naming convention are challenging to identify; thus, the coding process slows down, especially when dealing with a large codebase.
- **Increased Risk of Errors**: Inconsistent or unclear variable, property, and parameter names lead to misunderstandings about what the variable represents. This ambiguity leads to incorrect assumptions and, consequently, bugs in the code.
- **Collaboration Difficulties**: In a team setting, inconsistent naming conventions lead to confusion and miscommunication among team members.
- **Difficulty in Code Maintenance**: Inconsistent naming leads to an inconsistent codebase. The code is difficult to understand, and making changes feels like refactoring constantly, as you face different naming methods. Ultimately, it makes the codebase harder to maintain.

In summary, not adhering to a naming convention for variables, properties, and parameters can lead to confusion, errors, and inefficiencies, making the code harder to read, understand, and maintain.

how_to_fix

First, familiarize yourself with the particular naming convention of the project in question. Then, update the name to match the convention, as well as all usages of the name. For many IDEs, you can use built-in renaming and refactoring features to update all usages at once.

Noncompliant code example

With the default regular expression `^[_$A-Za-z][$A-Za-z0-9]*$|^[_$A-Z][_$A-Z0-9]+$`:

```
const foo_bar = 1; // Noncompliant
const baz_ = 2; // Noncompliant
```

Compliant solution

```
const fooBar = 1;
const _baz = 2;
```

Nested code blocks should not be used

Clave: javascript:S1199

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- Wikipedia - Single Responsibility Principle
- MDN web docs - block statement
- MDN web docs - var
- MDN web docs - let
- MDN web docs - const
- MDN web docs - class declaration
- MDN web docs - function declaration
- MDN web docs - strict mode

how_to_fix

The nested code blocks should be extracted into separate methods.

Noncompliant code example

```
{   // Noncompliant: redundant code block
    var foo = bar();
}

if (condition) {
    doSomething();
    {   // Noncompliant: redundant code block
        doOtherStuff();
    }
}
```

Compliant solution

```
var foo = bar();

if (condition) {
    doSomething();
    doOtherStuff();
}
```

root_cause

Nested code blocks can be used to create a new scope: variables declared within that block cannot be accessed from the outside, and their lifetime end at the end of the block. However, this only happens when you use ES6 `let` or `const` keywords, a class declaration or a function declaration (in strict mode). Otherwise, the nested block is redundant and should be removed.

Exceptions

The rule does not apply to the following cases:

- Block statements containing variable declarations using `let` or `const` keywords or class declarations are not redundant as they create a new scope.

```
{
    let x = 1;
}
```

- The same applies to function declarations in strict mode

```
"use strict";
{
    function foo() {}
}
```

- The rule also does not apply to the blocks that are part of the control flow.

```
if (condition) {
    doSomething();
}
```

Control structures should use curly braces

Clave: javascript:S121

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

While not technically incorrect, the omission of curly braces can be misleading and may lead to the introduction of errors during maintenance.

In the following example, the two calls seem to be attached to the `if` statement, but only the first one is, and `checkSomething` will always be executed:

```
if (condition)  // Noncompliant
  executeSomething();
  checkSomething();
```

Adding curly braces improves the code readability and its robustness:

```
if (condition) {
  executeSomething();
  checkSomething();
}
```

The rule raises an issue when a control structure has no curly braces.

introduction

Control structures are code statements that impact the program’s control flow (e.g., if statements, for loops, etc.)

Statements should be on separate lines

Clave: javascript:S122

Severidad: MAJOR

Estado: ⚠️ Depreciada

Descripción: No disponible

introduction

This rule is deprecated, and will eventually be removed.

root_cause

Putting multiple statements on a single line lowers the code readability and makes debugging the code more complex.

```
foo(); bar(); // Noncompliant
```

Write one statement per line to improve readability.

```
foo();
bar();
```

Hard-coded passwords are security-sensitive

Clave: javascript:S2068

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- Passwords allow access to a sensitive component like a database, a file storage, an API or a service.
- Passwords are used in production environments.
- Application re-distribution is required before updating the passwords.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host:'localhost',
  user: "admin",
  database: "project",
  password: "mypassword", // sensitive
  multipleStatements: true
});

connection.connect();
```

how_to_fix

Recommended Secure Coding Practices

- Store the passwords in a configuration file that is not pushed to the code repository.
- Store the passwords in a database.
- Use your cloud provider’s service for managing passwords.
- If a password has been disclosed through the source code: change it.

Compliant Solution

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host: process.env.MYSQL_URL,
  user: process.env.MYSQL_USERNAME,
  password: process.env.MYSQL_PASSWORD,
  database: process.env.MYSQL_DATABASE
});
connection.connect();
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-259 - Use of Hard-coded Password
- Derived from FindSecBugs rule Hard Coded Password

root_cause

Because it is easy to extract strings from an application source code or binary, passwords should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2019-13466
- CVE-2018-15389

Passwords should be stored outside of the code in a configuration file, a database, or a management service for passwords.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

default

Because it is easy to extract strings from an application source code or binary, passwords should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2019-13466
- CVE-2018-15389

Passwords should be stored outside of the code in a configuration file, a database, or a management service for passwords.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

Ask Yourself Whether

- Passwords allow access to a sensitive component like a database, a file storage, an API or a service.
- Passwords are used in production environments.
- Application re-distribution is required before updating the passwords.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Store the passwords in a configuration file that is not pushed to the code repository.
- Store the passwords in a database.

- Use your cloud provider’s service for managing passwords.
- If a password has been disclosed through the source code: change it.

Sensitive Code Example

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host:'localhost',
  user: "admin",
  database: "project",
  password: "mypassword", // sensitive
  multipleStatements: true
});

connection.connect();
```

Compliant Solution

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host: process.env.MYSQL_URL,
  user: process.env.MYSQL_USERNAME,
  password: process.env.MYSQL_PASSWORD,
  database: process.env.MYSQL_DATABASE
});
connection.connect();
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-259 - Use of Hard-coded Password
- Derived from FindSecBugs rule Hard Coded Password

Formatting SQL queries is security-sensitive

Clave: javascript:S2077

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn’t detect SQL injections (unlike rule S3649), the goal is only to highlight complex/formatted queries.

Exceptions

This rule’s current implementation does not follow variables. It will only detect SQL queries which are formatted directly in the function call.

```
const sql = 'SELECT * FROM users WHERE id = ' + userinput;
mycon.query(sql, (err, res) => {}); // Sensitive but no issue is raised.
```

assess_the_problem

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
// === MySQL ===
const mysql = require('mysql');
const mycon = mysql.createConnection({ host: host, user: user, password: pass, database: db });
mycon.connect(function(err) {
  mycon.query('SELECT * FROM users WHERE id = ' + userinput, (err, res) => {}); // Sensitive
});

// === PostgreSQL ===
const pg = require('pg');
const pgcon = new pg.Client({ host: host, user: user, password: pass, database: db });
pgcon.connect();
pgcon.query('SELECT * FROM users WHERE id = ' + userinput, (err, res) => {}); // Sensitive
```

default

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn’t detect SQL injections (unlike rule S3649), the goal is only to highlight complex/formatted queries.

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

Sensitive Code Example

```
// === MySQL ===
const mysql = require('mysql');
const mycon = mysql.createConnection({ host: host, user: user, password: pass, database: db });
mycon.connect(function(err) {
  mycon.query('SELECT * FROM users WHERE id = ' + userinput, (err, res) => {}); // Sensitive
});

// === PostgreSQL ===
const pg = require('pg');
const pgcon = new pg.Client({ host: host, user: user, password: pass, database: db });
pgcon.connect();
pgcon.query('SELECT * FROM users WHERE id = ' + userinput, (err, res) => {}); // Sensitive
```

Compliant Solution

```
// === MySQL ===
const mysql = require('mysql');
const mycon = mysql.createConnection({ host: host, user: user, password: pass, database: db });
mycon.connect(function(err) {
  mycon.query('SELECT name FROM users WHERE id = ?', [userinput], (err, res) => {});
});

// === PostgreSQL ===
const pg = require('pg');
const pgcon = new pg.Client({ host: host, user: user, password: pass, database: db });
pgcon.connect();
pgcon.query('SELECT name FROM users WHERE id = $1', [userinput], (err, res) => {});
```

Exceptions

This rule’s current implementation does not follow variables. It will only detect SQL queries which are formatted directly in the function call.

```
const sql = 'SELECT * FROM users WHERE id = ' + userinput;
mycon.query(sql, (err, res) => {}); // Sensitive but no issue is raised.
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

how_to_fix

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

Compliant Solution

```
// === MySQL ===
const mysql = require('mysql');
const mycon = mysql.createConnection({ host: host, user: user, password: pass, database: db });
mycon.connect(function(err) {
  mycon.query('SELECT name FROM users WHERE id = ?', [userinput], (err, res) => {});
});

// === PostgreSQL ===
const pg = require('pg');
const pgcon = new pg.Client({ host: host, user: user, password: pass, database: db });
pgcon.connect();
pgcon.query('SELECT name FROM users WHERE id = $1', [userinput], (err, res) => {});
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Test files should contain at least one test case

Clave: javascript:S2187

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

Test files in JavaScript and TypeScript are meant to contain test cases. These test cases are used to verify the functionality of your code and ensure that it behaves as expected. If a test file doesn’t contain any test cases, it’s not serving its purpose.

A test file without test cases might indicate:

- An incomplete test suite: Perhaps the developer started writing tests but didn’t finish.
- A mistake: The developer might have accidentally deleted the test cases or moved them to another file.

This rule flags any file that has `.test` or `.spec` as part of its suffix but does not contain any test cases defined using the different forms of the `it` and `test` functions from Jasmine, Jest, Mocha, or Node.js testing API.

resources

Documentation

- Jasmine docs - API
- Jest docs - API
- Mocha docs - API
- Node.js docs - API

how_to_fix

Add test cases to the file or delete it if it isn’t needed anymore.

Noncompliant code example

```
// eval.test.js

/* no test cases */
```

Compliant solution

```
// eval.test.js

it('1 + 2 should give 3', () => {
    expect(1 + 2).toBe(3);
});
```

Loops should not be infinite

Clave: javascript:S2189

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Loops and iteration
- MDN web docs - `for`
- MDN web docs - `while`
- MDN web docs - `do...while`
- MDN web docs - `break`
- Wikipedia - Infinite loop

root_cause

A loop is a control structure that allows a block of code to be executed repeatedly until a certain condition is met. The basic idea behind a loop is to automate repetitive tasks, such as iterating over a collection of data or performing a calculation multiple times with different inputs.

An infinite loop is a loop that runs indefinitely without ever terminating. In other words, the loop condition is always true, and the loop never exits. This can happen when the loop condition is not defined or when the loop condition is never met.

Infinite loops can cause a program to hang or crash, as the program will continue to execute the loop indefinitely.

This rule will raise an issue on `for`, `while` and `do...while` loops where no clear exit condition has been found.

There are some known limitations for this rule:

- False positives: when an exception is raised by a function invoked within the loop.
- False negatives: when a loop condition is based on an element of an array or object.

```
for (;;) {  // Noncompliant: end condition omitted
  // ...
}

let j = 0;
while (true) { // Noncompliant: constant end condition
  j++;
}

let k;
let b = true;
while (b) { // Noncompliant: constant end condition
  k++;
}
```

Ensure the loop condition is defined or use a `break` statement.

```
for (let i = 0; i < 5; i++) {
  // ...
}

let j = 0;
while (true) {
  j++;
  if (j < 5) {
    break;
  }
}

let k;
let b = true;
while (b) {
  k++;
  b = k < 10;
}
```

Object literal shorthand syntax should be used

Clave: javascript:S3498

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Object initializer
- MDN web docs - Property definitions
- MDN web docs - Method definitions

root_cause

In JavaScript, object shorthand syntax is a more concise way to define properties on objects. It was introduced to make object literals more readable and expressive.

In the shorthand syntax, if a variable exists in the scope with the same name as the object key you’re defining, you can omit the key-value pair and just write the variable name. The interpreter will automatically understand that the key and the variable are linked.

Using object shorthand syntax can make your code cleaner and easier to read. It can also reduce the chance of making errors, as you don’t have to repeat yourself by writing the variable name twice.

```
let a = 1;

let myObj = {
  a : a,  // Noncompliant
  fun: function () {  // Noncompliant
    //...
  }
}
```

You can omit the property name and the colon if it is the same as the local variable name. Similarly, you can omit the `function` keyword for method definitions.

```
let a = 1;

let myObj = {
  a,
  fun () {
    //...
  }
}
```

Shorthand object properties should be grouped at the beginning or end of an object declaration

Clave: javascript:S3499

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Grouping all the shorthand declarations together in an object makes the declaration as a whole more readable. This rule accepts shorthand declarations grouped at either the beginning or end of an object.

Noncompliant code example

```
let obj1 = {
  foo,
  a: 1,
  color,  // Noncompliant
  b: 2,
  judyGarland  // Noncompliant
}
```

Compliant solution

```
let obj1 = {
  foo,
  color,
  judyGarland,
  a: 1,
  b: 2
}
```

or

```
let obj1 = {
  a: 1,
  b: 2,
  foo,
  color,
  judyGarland
}
```

"typeof" expressions should only be compared to valid values

Clave: javascript:S4125

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - typeof
- MDN web docs - Primitive values

root_cause

The `typeof` operator returns a string indicating the type of its argument, and the set of returned values is limited:

- `"undefined"`
- `"boolean"`
- `"number"`
- `"string"`
- `"symbol"` (since ECMAScript 2015)
- `"function"`
- `"object"` (for `null` and any other object)
- `"bigint"` (since ECMAScript 2020)

Compare a `typeof` expression to anything else, and the result will always be `false`.

```
function isNumber(x) {
  return typeof x === "Number"; // Noncompliant: the function always returns 'false'
}
```

Instead, make sure you are always comparing the expression against one of the seven possible values.

```
function isNumber(x) {
  return typeof x === "number";
}
```

"for of" should be used with Iterables

Clave: javascript:S4138

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

`for...of` statements are used to iterate over the values of an iterable object. Iterables are objects implementing the `@@iterator` method, which returns an object conforming to the iterator protocol. JavaScript provides many built-in iterables that can and should be used with this looping statement.

The use of the `for...of` statement is recommended over the `for` statement when iterating through iterable objects as simplifies the syntax and eliminates the need for a counter variable.

```
const arr = [4, 3, 2, 1];

for (let i = 0; i < arr.length; i++) {  // Noncompliant: arr is an iterable object
  console.log(arr[i]);
}
```

When looping over an iterable, use the `for...of` for better readability.

```
const arr = [4, 3, 2, 1];

for (let value of arr) {
  console.log(value);
}
```

resources

Documentation

- MDN web docs - `for...of`
- MDN web docs - Iterator protocol

"for in" should not be used with iterables

Clave: javascript:S4139

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

If you have an iterable, such as an array, set, or list, your best option for looping through its values is the `for of` syntax. Use `for in` and you'll iterate the properties, rather than the values.

Noncompliant code example

```
const arr = [4, 3, 2, 1];

for (let value in arr) {  // Noncompliant
  console.log(value);  // logs 0, 1, 2, 3
}
```

Compliant solution

```
const arr = [4, 3, 2, 1];

for (let value of arr) {
  console.log(value);
}
```

Using clear-text protocols is security-sensitive

Clave: javascript:S5332

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- Application data needs to be protected against falsifications or leaks when transiting over the network.
- Application data transits over an untrusted network.
- Compliance rules require the service to encrypt data in transit.
- Your application renders web pages with a relaxed mixed content policy.
- OS-level protections against clear-text traffic are deactivated.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
url = "http://example.com"; // Sensitive
url = "ftp://anonymous@example.com"; // Sensitive
url = "telnet://anonymous@example.com"; // Sensitive
```

For nodemailer:

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({
  secure: false, // Sensitive
  requireTLS: false // Sensitive
});
```

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({}); // Sensitive
```

For ftp:

```
var Client = require('ftp');
var c = new Client();
c.connect({
  'secure': false // Sensitive
});
```

For telnet-client:

```
const Telnet = require('telnet-client'); // Sensitive
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationLoadBalancer:

```
import { ApplicationLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const alb = new ApplicationLoadBalancer(this, 'ALB', {
  vpc: vpc,
  internetFacing: true
});

alb.addListener('listener-http-default', {
  port: 8080,
  open: true
}); // Sensitive

alb.addListener('listener-http-explicit', {
  protocol: ApplicationProtocol.HTTP, // Sensitive
  port: 8080,
  open: true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationListener:

```
import { ApplicationListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new ApplicationListener(this, 'listener-http-explicit-constructor', {
  loadBalancer: alb,
  protocol: ApplicationProtocol.HTTP, // Sensitive
  port: 8080,
  open: true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkLoadBalancer:

```
import { NetworkLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const nlb = new NetworkLoadBalancer(this, 'nlb', {
  vpc: vpc,
  internetFacing: true
});

var listenerNLB = nlb.addListener('listener-tcp-default', {
  port: 1234
}); // Sensitive

listenerNLB = nlb.addListener('listener-tcp-explicit', {
  protocol: Protocol.TCP, // Sensitive
  port: 1234
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkListener:

```
import { NetworkListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new NetworkListener(this, 'listener-tcp-explicit-constructor', {
  loadBalancer: nlb,
  protocol: Protocol.TCP, // Sensitive
  port: 8080
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.CfnListener:

```
import { CfnListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new CfnListener(this, 'listener-http', {
  defaultActions: defaultActions,
  loadBalancerArn: alb.loadBalancerArn,
  protocol: "HTTP", // Sensitive
  port: 80
});

new CfnListener(this, 'listener-tcp', {
  defaultActions: defaultActions,
  loadBalancerArn: alb.loadBalancerArn,
  protocol: "TCP", // Sensitive
  port: 80
});
```

For aws-cdk-lib.aws-elasticloadbalancing.CfnLoadBalancer:

```
import { CfnLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

new CfnLoadBalancer(this, 'elb-tcp', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'tcp' // Sensitive
  }]
});

new CfnLoadBalancer(this, 'elb-http', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'http' // Sensitive
  }]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.LoadBalancer:

```
import { LoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

const loadBalancer = new LoadBalancer(this, 'elb-tcp-dict', {
    vpc,
    internetFacing: true,
    healthCheck: {
    port: 80,
    },
    listeners: [
    {
        externalPort:10000,
        externalProtocol: LoadBalancingProtocol.TCP, // Sensitive
        internalPort:10000
    }]
});

loadBalancer.addListener({
  externalPort:10001,
  externalProtocol:LoadBalancingProtocol.TCP, // Sensitive
  internalPort:10001
});
loadBalancer.addListener({
  externalPort:10002,
  externalProtocol:LoadBalancingProtocol.HTTP, // Sensitive
  internalPort:10002
});
```

For aws-cdk-lib.aws-elasticache.CfnReplicationGroup:

```
import { CfnReplicationGroup } from 'aws-cdk-lib/aws-elasticache';

new CfnReplicationGroup(this, 'unencrypted-implicit', {
  replicationGroupDescription: 'exampleDescription'
}); // Sensitive

new CfnReplicationGroup(this, 'unencrypted-explicit', {
  replicationGroupDescription: 'exampleDescription',
  transitEncryptionEnabled: false // Sensitive
});
```

For aws-cdk-lib.aws-kinesis.CfnStream:

```
import { CfnStream } from 'aws-cdk-lib/aws-kinesis';

new CfnStream(this, 'cfnstream-implicit-unencrytped', undefined); // Sensitive

new CfnStream(this, 'cfnstream-explicit-unencrytped', {
  streamEncryption: undefined // Sensitive
});
```

For aws-cdk-lib.aws-kinesis.Stream:

```
import { Stream } from 'aws-cdk-lib/aws-kinesis';

new Stream(this, 'stream-explicit-unencrypted', {
  encryption: StreamEncryption.UNENCRYPTED // Sensitive
});
```

how_to_fix

Recommended Secure Coding Practices

- Make application data transit over a secure, authenticated and encrypted protocol like TLS or SSH. Here are a few alternatives to the most common clear-text protocols:
  - Use `ssh` as an alternative to `telnet`.
  - Use `sftp`, `scp`, or `ftps` instead of `ftp`.
  - Use `https` instead of `http`.
  - Use `SMTP` over `SSL/TLS` or `SMTP` with `STARTTLS` instead of clear-text SMTP.
- Enable encryption of cloud components communications whenever it is possible.
- Configure your application to block mixed content when rendering web pages.
- If available, enforce OS-level deactivation of all clear-text traffic.

It is recommended to secure all transport channels, even on local networks, as it can take a single non-secure connection to compromise an entire application or system.

Compliant Solution

```
url = "https://example.com";
url = "sftp://anonymous@example.com";
url = "ssh://anonymous@example.com";
```

For nodemailer one of the following options must be set:

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({
  secure: true,
  requireTLS: true,
  port: 465,
  secured: true
});
```

For ftp:

```
var Client = require('ftp');
var c = new Client();
c.connect({
  'secure': true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationLoadBalancer:

```
import { ApplicationLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const alb = new ApplicationLoadBalancer(this, 'ALB', {
  vpc: vpc,
  internetFacing: true
});

alb.addListener('listener-https-explicit', {
  protocol: ApplicationProtocol.HTTPS,
  port: 8080,
  open: true,
  certificates: [certificate]
});

alb.addListener('listener-https-implicit', {
  port: 8080,
  open: true,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationListener:

```
import { ApplicationListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new ApplicationListener(this, 'listener-https-explicit', {
  loadBalancer: loadBalancer,
  protocol: ApplicationProtocol.HTTPS,
  port: 8080,
  open: true,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkLoadBalancer:

```
import { NetworkLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const nlb = new NetworkLoadBalancer(this, 'nlb', {
  vpc: vpc,
  internetFacing: true
});

nlb.addListener('listener-tls-explicit', {
  protocol: Protocol.TLS,
  port: 1234,
  certificates: [certificate]
});

nlb.addListener('listener-tls-implicit', {
  port: 1234,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkListener:

```
import { NetworkListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new NetworkListener(this, 'listener-tls-explicit', {
  loadBalancer: loadBalancer,
  protocol: Protocol.TLS,
  port: 8080,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.CfnListener:

```
import { CfnListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new CfnListener(this, 'listener-https', {
  defaultActions: defaultActions,
  loadBalancerArn: loadBalancerArn,
  protocol: "HTTPS",
  port: 80
  certificates: [certificate]
});

new CfnListener(this, 'listener-tls', {
  defaultActions: defaultActions,
  loadBalancerArn: loadBalancerArn,
  protocol: "TLS",
  port: 80
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.CfnLoadBalancer:

```
import { CfnLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

new CfnLoadBalancer(this, 'elb-ssl', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'ssl',
    sslCertificateId: sslCertificateId
  }]
});

new CfnLoadBalancer(this, 'elb-https', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'https',
    sslCertificateId: sslCertificateId
  }]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.LoadBalancer:

```
import { LoadBalancer, LoadBalancingProtocol } from 'aws-cdk-lib/aws-elasticloadbalancing';

const lb = new LoadBalancer(this, 'elb-ssl', {
  vpc,
  internetFacing: true,
  healthCheck: {
    port: 80,
  },
  listeners: [
    {
      externalPort:10000,
      externalProtocol:LoadBalancingProtocol.SSL,
      internalPort:10000
    }]
});

lb.addListener({
  externalPort:10001,
  externalProtocol:LoadBalancingProtocol.SSL,
  internalPort:10001
});
lb.addListener({
  externalPort:10002,
  externalProtocol:LoadBalancingProtocol.HTTPS,
  internalPort:10002
});
```

For aws-cdk-lib.aws-elasticache.CfnReplicationGroup:

```
import { CfnReplicationGroup } from 'aws-cdk-lib/aws-elasticache';

new CfnReplicationGroup(this, 'encrypted-explicit', {
  replicationGroupDescription: 'example',
  transitEncryptionEnabled: true
});
```

For aws-cdk-lib.aws-kinesis.Stream:

```
import { Stream } from 'aws-cdk-lib/aws-kinesis';

new Stream(this, 'stream-implicit-encrypted');

new Stream(this, 'stream-explicit-encrypted-selfmanaged', {
  encryption: StreamEncryption.KMS,
  encryptionKey: encryptionKey,
});

new Stream(this, 'stream-explicit-encrypted-managed', {
  encryption: StreamEncryption.MANAGED
});
```

For aws-cdk-lib.aws-kinesis.CfnStream:

```
import { CfnStream } from 'aws-cdk-lib/aws-kinesis';

new CfnStream(this, 'cfnstream-explicit-encrypted', {
  streamEncryption: {
    encryptionType: encryptionType,
    keyId: encryptionKey.keyId,
  }
});
```

See

Documentation

- AWS Documentation - Listeners for your Application Load Balancers
- AWS Documentation - Stream Encryption

Articles & blog posts

- Google - Moving towards more secure web
- Mozilla - Deprecating non secure http

Standards

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information
- STIG Viewer - Application Security and Development: V-222397 - The application must implement cryptographic mechanisms to protect the integrity of remote access sessions.
- STIG Viewer - Application Security and Development: V-222534 - Service-Oriented Applications handling non-releasable data must authenticate endpoint devices via mutual SSL/TLS.
- STIG Viewer - Application Security and Development: V-222562 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the integrity of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222563 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the confidentiality of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222577 - The application must not expose session IDs.
- STIG Viewer - Application Security and Development: V-222596 - The application must protect the confidentiality and integrity of transmitted information.
- STIG Viewer - Application Security and Development: V-222597 - The application must implement cryptographic mechanisms to prevent unauthorized disclosure of information and/or detect changes to information during transmission.
- STIG Viewer - Application Security and Development: V-222598 - The application must maintain the confidentiality and integrity of information during preparation for transmission.
- STIG Viewer - Application Security and Development: V-222599 - The application must maintain the confidentiality and integrity of information during reception.

default

Clear-text protocols such as `ftp`, `telnet`, or `http` lack encryption of transported data, as well as the capability to build an authenticated connection. It means that an attacker able to sniff traffic from the network can read, modify, or corrupt the transported content. These protocols are not secure as they expose applications to an extensive range of risks:

- sensitive data exposure
- traffic redirected to a malicious endpoint
- malware-infected software update or installer
- execution of client-side code
- corruption of critical information

Even in the context of isolated networks like offline environments or segmented cloud environments, the insider threat exists. Thus, attacks involving communications being sniffed or tampered with can still happen.

For example, attackers could successfully compromise prior security layers by:

- bypassing isolation mechanisms
- compromising a component of the network
- getting the credentials of an internal IAM account (either from a service account or an actual person)

In such cases, encrypting communications would decrease the chances of attackers to successfully leak data or steal credentials from other network components. By layering various security practices (segmentation and encryption, for example), the application will follow the *defense-in-depth* principle.

Note that using the `http` protocol is being deprecated by major web browsers.

In the past, it has led to the following vulnerabilities:

- CVE-2019-6169
- CVE-2019-12327
- CVE-2019-11065

Ask Yourself Whether

- Application data needs to be protected against falsifications or leaks when transiting over the network.
- Application data transits over an untrusted network.
- Compliance rules require the service to encrypt data in transit.

- Your application renders web pages with a relaxed mixed content policy.
- OS-level protections against clear-text traffic are deactivated.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Make application data transit over a secure, authenticated and encrypted protocol like TLS or SSH. Here are a few alternatives to the most common clear-text protocols:
  - Use `ssh` as an alternative to `telnet`.
  - Use `sftp`, `scp`, or `ftps` instead of `ftp`.
  - Use `https` instead of `http`.
  - Use `SMTP` over `SSL/TLS` or `SMTP` with `STARTTLS` instead of clear-text SMTP.
- Enable encryption of cloud components communications whenever it is possible.
- Configure your application to block mixed content when rendering web pages.
- If available, enforce OS-level deactivation of all clear-text traffic.

It is recommended to secure all transport channels, even on local networks, as it can take a single non-secure connection to compromise an entire application or system.

Sensitive Code Example

```
url = "http://example.com"; // Sensitive
url = "ftp://anonymous@example.com"; // Sensitive
url = "telnet://anonymous@example.com"; // Sensitive
```

For nodemailer:

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({
  secure: false, // Sensitive
  requireTLS: false // Sensitive
});
```

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({}); // Sensitive
```

For ftp:

```
var Client = require('ftp');
var c = new Client();
c.connect({
  'secure': false // Sensitive
});
```

For telnet-client:

```
const Telnet = require('telnet-client'); // Sensitive
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationLoadBalancer:

```
import { ApplicationLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const alb = new ApplicationLoadBalancer(this, 'ALB', {
  vpc: vpc,
  internetFacing: true
});

alb.addListener('listener-http-default', {
  port: 8080,
  open: true
}); // Sensitive

alb.addListener('listener-http-explicit', {
  protocol: ApplicationProtocol.HTTP, // Sensitive
  port: 8080,
  open: true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationListener:

```
import { ApplicationListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new ApplicationListener(this, 'listener-http-explicit-constructor', {
  loadBalancer: alb,
  protocol: ApplicationProtocol.HTTP, // Sensitive
  port: 8080,
  open: true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkLoadBalancer:

```
import { NetworkLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const nlb = new NetworkLoadBalancer(this, 'nlb', {
  vpc: vpc,
  internetFacing: true
});
```

```
var listenerNLB = nlb.addListener('listener-tcp-default', {
  port: 1234
}); // Sensitive

listenerNLB = nlb.addListener('listener-tcp-explicit', {
  protocol: Protocol.TCP, // Sensitive
  port: 1234
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkListener:

```
import { NetworkListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new NetworkListener(this, 'listener-tcp-explicit-constructor', {
  loadBalancer: nlb,
  protocol: Protocol.TCP, // Sensitive
  port: 8080
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.CfnListener:

```
import { CfnListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new CfnListener(this, 'listener-http', {
  defaultActions: defaultActions,
  loadBalancerArn: alb.loadBalancerArn,
  protocol: "HTTP", // Sensitive
  port: 80
});

new CfnListener(this, 'listener-tcp', {
  defaultActions: defaultActions,
  loadBalancerArn: alb.loadBalancerArn,
  protocol: "TCP", // Sensitive
  port: 80
});
```

For aws-cdk-lib.aws-elasticloadbalancing.CfnLoadBalancer:

```
import { CfnLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

new CfnLoadBalancer(this, 'elb-tcp', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'tcp' // Sensitive
  }]
});

new CfnLoadBalancer(this, 'elb-http', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'http' // Sensitive
  }]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.LoadBalancer:

```
import { LoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

const loadBalancer = new LoadBalancer(this, 'elb-tcp-dict', {
    vpc,
    internetFacing: true,
    healthCheck: {
    port: 80,
    },
    listeners: [
    {
        externalPort:10000,
        externalProtocol: LoadBalancingProtocol.TCP, // Sensitive
        internalPort:10000
    }]
});

loadBalancer.addListener({
  externalPort:10001,
  externalProtocol:LoadBalancingProtocol.TCP, // Sensitive
  internalPort:10001
});
loadBalancer.addListener({
  externalPort:10002,
  externalProtocol:LoadBalancingProtocol.HTTP, // Sensitive
  internalPort:10002
});
```

For aws-cdk-lib.aws-elasticache.CfnReplicationGroup:

```
import { CfnReplicationGroup } from 'aws-cdk-lib/aws-elasticache';

new CfnReplicationGroup(this, 'unencrypted-implicit', {
  replicationGroupDescription: 'exampleDescription'
}); // Sensitive
```

```
new CfnReplicationGroup(this, 'unencrypted-explicit', {
  replicationGroupDescription: 'exampleDescription',
  transitEncryptionEnabled: false // Sensitive
});
```

For aws-cdk-lib.aws-kinesis.CfnStream:

```
import { CfnStream } from 'aws-cdk-lib/aws-kinesis';

new CfnStream(this, 'cfnstream-implicit-unencrytped', undefined); // Sensitive

new CfnStream(this, 'cfnstream-explicit-unencrytped', {
  streamEncryption: undefined // Sensitive
});
```

For aws-cdk-lib.aws-kinesis.Stream:

```
import { Stream } from 'aws-cdk-lib/aws-kinesis';

new Stream(this, 'stream-explicit-unencrypted', {
  encryption: StreamEncryption.UNENCRYPTED // Sensitive
});
```

Compliant Solution

```
url = "https://example.com";
url = "sftp://anonymous@example.com";
url = "ssh://anonymous@example.com";
```

For nodemailer one of the following options must be set:

```
const nodemailer = require("nodemailer");
let transporter = nodemailer.createTransport({
  secure: true,
  requireTLS: true,
  port: 465,
  secured: true
});
```

For ftp:

```
var Client = require('ftp');
var c = new Client();
c.connect({
  'secure': true
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationLoadBalancer:

```
import { ApplicationLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const alb = new ApplicationLoadBalancer(this, 'ALB', {
  vpc: vpc,
  internetFacing: true
});

alb.addListener('listener-https-explicit', {
  protocol: ApplicationProtocol.HTTPS,
  port: 8080,
  open: true,
  certificates: [certificate]
});

alb.addListener('listener-https-implicit', {
  port: 8080,
  open: true,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.ApplicationListener:

```
import { ApplicationListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new ApplicationListener(this, 'listener-https-explicit', {
  loadBalancer: loadBalancer,
  protocol: ApplicationProtocol.HTTPS,
  port: 8080,
  open: true,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkLoadBalancer:

```
import { NetworkLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

const nlb = new NetworkLoadBalancer(this, 'nlb', {
  vpc: vpc,
  internetFacing: true
});

nlb.addListener('listener-tls-explicit', {
  protocol: Protocol.TLS,
  port: 1234,
  certificates: [certificate]
});

nlb.addListener('listener-tls-implicit', {
  port: 1234,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.NetworkListener:

```
import { NetworkListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new NetworkListener(this, 'listener-tls-explicit', {
  loadBalancer: loadBalancer,
  protocol: Protocol.TLS,
  port: 8080,
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancingv2.CfnListener:

```
import { CfnListener } from 'aws-cdk-lib/aws-elasticloadbalancingv2';

new CfnListener(this, 'listener-https', {
  defaultActions: defaultActions,
  loadBalancerArn: loadBalancerArn,
  protocol: "HTTPS",
  port: 80
  certificates: [certificate]
});

new CfnListener(this, 'listener-tls', {
  defaultActions: defaultActions,
  loadBalancerArn: loadBalancerArn,
  protocol: "TLS",
  port: 80
  certificates: [certificate]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.CfnLoadBalancer:

```
import { CfnLoadBalancer } from 'aws-cdk-lib/aws-elasticloadbalancing';

new CfnLoadBalancer(this, 'elb-ssl', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'ssl',
    sslCertificateId: sslCertificateId
  }]
});

new CfnLoadBalancer(this, 'elb-https', {
  listeners: [{
    instancePort: '1000',
    loadBalancerPort: '1000',
    protocol: 'https',
    sslCertificateId: sslCertificateId
  }]
});
```

For aws-cdk-lib.aws-elasticloadbalancing.LoadBalancer:

```
import { LoadBalancer, LoadBalancingProtocol } from 'aws-cdk-lib/aws-elasticloadbalancing';

const lb = new LoadBalancer(this, 'elb-ssl', {
  vpc,
  internetFacing: true,
  healthCheck: {
    port: 80,
  },
  listeners: [
    {
      externalPort:10000,
      externalProtocol:LoadBalancingProtocol.SSL,
      internalPort:10000
    }]
});

lb.addListener({
  externalPort:10001,
  externalProtocol:LoadBalancingProtocol.SSL,
  internalPort:10001
});
lb.addListener({
  externalPort:10002,
  externalProtocol:LoadBalancingProtocol.HTTPS,
  internalPort:10002
});
```

For aws-cdk-lib.aws-elasticache.CfnReplicationGroup:

```
import { CfnReplicationGroup } from 'aws-cdk-lib/aws-elasticache';

new CfnReplicationGroup(this, 'encrypted-explicit', {
  replicationGroupDescription: 'example',
  transitEncryptionEnabled: true
});
```

For aws-cdk-lib.aws-kinesis.Stream:

```
import { Stream } from 'aws-cdk-lib/aws-kinesis';

new Stream(this, 'stream-implicit-encrypted');

new Stream(this, 'stream-explicit-encrypted-selfmanaged', {
  encryption: StreamEncryption.KMS,
  encryptionKey: encryptionKey,
});

new Stream(this, 'stream-explicit-encrypted-managed', {
  encryption: StreamEncryption.MANAGED
});
```

For aws-cdk-lib.aws-kinesis.CfnStream:

```
import { CfnStream } from 'aws-cdk-lib/aws-kinesis';

new CfnStream(this, 'cfnstream-explicit-encrypted', {
  streamEncryption: {
    encryptionType: encryptionType,
    keyId: encryptionKey.keyId,
  }
});
```

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Insecure protocol scheme followed by loopback addresses like 127.0.0.1 or `localhost`.

See

Documentation

- AWS Documentation - Listeners for your Application Load Balancers
- AWS Documentation - Stream Encryption

Articles & blog posts

- Google - Moving towards more secure web
- Mozilla - Deprecating non secure http

Standards

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information
- STIG Viewer - Application Security and Development: V-222397 - The application must implement cryptographic mechanisms to protect the integrity of remote access sessions.
- STIG Viewer - Application Security and Development: V-222534 - Service-Oriented Applications handling non-releasable data must authenticate endpoint devices via mutual SSL/TLS.
- STIG Viewer - Application Security and Development: V-222562 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the integrity of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222563 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the confidentiality of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222577 - The application must not expose session IDs.
- STIG Viewer - Application Security and Development: V-222596 - The application must protect the confidentiality and integrity of transmitted information.
- STIG Viewer - Application Security and Development: V-222597 - The application must implement cryptographic mechanisms to prevent unauthorized disclosure of information and/or detect changes to information during transmission.
- STIG Viewer - Application Security and Development: V-222598 - The application must maintain the confidentiality and integrity of information during preparation for transmission.
- STIG Viewer - Application Security and Development: V-222599 - The application must maintain the confidentiality and integrity of information during reception.

root_cause

Clear-text protocols such as `ftp`, `telnet`, or `http` lack encryption of transported data, as well as the capability to build an authenticated connection. It means that an attacker able to sniff traffic from the network can read, modify, or corrupt the transported content. These protocols are not secure as they expose applications to an extensive range of risks:

- sensitive data exposure
- traffic redirected to a malicious endpoint
- malware-infected software update or installer

- execution of client-side code
- corruption of critical information

Even in the context of isolated networks like offline environments or segmented cloud environments, the insider threat exists. Thus, attacks involving communications being sniffed or tampered with can still happen.

For example, attackers could successfully compromise prior security layers by:

- bypassing isolation mechanisms
- compromising a component of the network
- getting the credentials of an internal IAM account (either from a service account or an actual person)

In such cases, encrypting communications would decrease the chances of attackers to successfully leak data or steal credentials from other network components. By layering various security practices (segmentation and encryption, for example), the application will follow the *defense-in-depth* principle.

Note that using the `http` protocol is being deprecated by major web browsers.

In the past, it has led to the following vulnerabilities:

- CVE-2019-6169
- CVE-2019-12327
- CVE-2019-11065

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Insecure protocol scheme followed by loopback addresses like 127.0.0.1 or `localhost`.

Statically serving hidden files is security-sensitive

Clave: javascript:S5691

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Hidden files are created automatically by many tools to save user-preferences, well-known examples are `.profile`, `.bashrc`, `.bash_history` or `.git`. To simplify the view these files are not displayed by default using operating system commands like `ls`.

Outside of the user environment, hidden files are sensitive because they are used to store privacy-related information or even hard-coded secrets.

Ask Yourself Whether

- Hidden files may have been inadvertently uploaded to the static server’s public directory and it accepts requests to hidden files.
- There is no business use cases linked to serve files in `.name` format but the server is not configured to reject requests to this type of files.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Disable the serving of hidden files.

Sensitive Code Example

Express.js serve-static middleware:

```
let serveStatic = require("serve-static");
let app = express();
let serveStaticMiddleware = serveStatic('public', { 'index': false, 'dotfiles': 'allow'});   // Sensitive
app.use(serveStaticMiddleware);
```

Compliant Solution

Express.js serve-static middleware:

```
let serveStatic = require("serve-static");
let app = express();
let serveStaticMiddleware = serveStatic('public', { 'index': false, 'dotfiles': 'ignore'});   // Compliant: ignore or deny are recommended values
let serveStaticDefault = serveStatic('public', { 'index': false});   // Compliant: by default, "dotfiles" (file or directory that begins with a dot) are
app.use(serveStaticMiddleware);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- github.com/mtojek/go-url-fuzzer - Discover hidden files and directories on a web server.
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-538 - File and Directory Information Exposure

assess_the_problem

Ask Yourself Whether

- Hidden files may have been inadvertently uploaded to the static server's public directory and it accepts requests to hidden files.
- There is no business use cases linked to serve files in `.name` format but the server is not configured to reject requests to this type of files.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Express.js serve-static middleware:

```
let serveStatic = require("serve-static");
let app = express();
let serveStaticMiddleware = serveStatic('public', { 'index': false, 'dotfiles': 'allow'});   // Sensitive
app.use(serveStaticMiddleware);
```

how_to_fix

Recommended Secure Coding Practices

- Disable the serving of hidden files.

Compliant Solution

Express.js serve-static middleware:

```
let serveStatic = require("serve-static");
let app = express();
let serveStaticMiddleware = serveStatic('public', { 'index': false, 'dotfiles': 'ignore'});   // Compliant: ignore or deny are recommended values
let serveStaticDefault = serveStatic('public', { 'index': false});   // Compliant: by default, "dotfiles" (file or directory that begins with a dot) are
app.use(serveStaticMiddleware);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- github.com/mtojek/go-url-fuzzer - Discover hidden files and directories on a web server.
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-538 - File and Directory Information Exposure

root_cause

Hidden files are created automatically by many tools to save user-preferences, well-known examples are `.profile`, `.bashrc`, `.bash_history` or `.git`. To simplify the view these files are not displayed by default using operating system commands like `ls`.

Outside of the user environment, hidden files are sensitive because they are used to store privacy-related information or even hard-coded secrets.

Allowing requests with excessive content length is security-sensitive

Clave: javascript:S5693

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- size limits are not defined for the different resources of the web application.

- the web application is not protected by rate limiting features.
- the web application infrastructure has limited resources.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

formidable file upload module:

```
const form = new Formidable();
form.maxFileSize = 10000000; // Sensitive: 10MB is more than the recommended limit of 8MB

const formDefault = new Formidable(); // Sensitive, the default value is 200MB
```

multer (Express.js middleware) file upload module:

```
let diskUpload = multer({
  storage: diskStorage,
  limits: {
    fileSize: 10000000; // Sensitive: 10MB is more than the recommended limit of 8MB
  }
});

let diskUploadUnlimited = multer({ // Sensitive: the default value is no limit
  storage: diskStorage,
});
```

body-parser module:

```
// 4MB is more than the recommended limit of 2MB for non-file-upload requests
let jsonParser = bodyParser.json({ limit: "4mb" }); // Sensitive
let urlencodedParser = bodyParser.urlencoded({ extended: false, limit: "4mb" }); // Sensitive
```

root_cause

Rejecting requests with significant content length is a good practice to control the network traffic intensity and thus resource consumption in order to prevent DoS attacks.

default

Rejecting requests with significant content length is a good practice to control the network traffic intensity and thus resource consumption in order to prevent DoS attacks.

Ask Yourself Whether

- size limits are not defined for the different resources of the web application.
- the web application is not protected by rate limiting features.
- the web application infrastructure has limited resources.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- For most of the features of an application, it is recommended to limit the size of requests to:
  - lower or equal to 8mb for file uploads.
  - lower or equal to 2mb for other requests.

It is recommended to customize the rule with the limit values that correspond to the web application.

Sensitive Code Example

formidable file upload module:

```
const form = new Formidable();
form.maxFileSize = 10000000; // Sensitive: 10MB is more than the recommended limit of 8MB

const formDefault = new Formidable(); // Sensitive, the default value is 200MB
```

multer (Express.js middleware) file upload module:

```
let diskUpload = multer({
  storage: diskStorage,
  limits: {
    fileSize: 10000000; // Sensitive: 10MB is more than the recommended limit of 8MB
  }
});

let diskUploadUnlimited = multer({ // Sensitive: the default value is no limit
  storage: diskStorage,
});
```

body-parser module:

```
// 4MB is more than the recommended limit of 2MB for non-file-upload requests
let jsonParser = bodyParser.json({ limit: "4mb" }); // Sensitive
let urlencodedParser = bodyParser.urlencoded({ extended: false, limit: "4mb" }); // Sensitive
```

Compliant Solution

formidable file upload module:

```
const form = new Formidable();
form.maxFileSize = 8000000; // Compliant: 8MB
```

multer (Express.js middleware) file upload module:

```
let diskUpload = multer({
  storage: diskStorage,
  limits: {
     fileSize: 8000000 // Compliant: 8MB
  }
});
```

body-parser module:

```
let jsonParser = bodyParser.json(); // Compliant, when the limit is not defined, the default value is set to 100kb
let urlencodedParser = bodyParser.urlencoded({ extended: false, limit: "2mb" }); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Owasp Cheat Sheet - Owasp Denial of Service Cheat Sheet
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-770 - Allocation of Resources Without Limits or Throttling
- CWE - CWE-400 - Uncontrolled Resource Consumption

how_to_fix

Recommended Secure Coding Practices

- For most of the features of an application, it is recommended to limit the size of requests to:
  - lower or equal to 8mb for file uploads.
  - lower or equal to 2mb for other requests.

It is recommended to customize the rule with the limit values that correspond to the web application.

Compliant Solution

formidable file upload module:

```
const form = new Formidable();
form.maxFileSize = 8000000; // Compliant: 8MB
```

multer (Express.js middleware) file upload module:

```
let diskUpload = multer({
  storage: diskStorage,
  limits: {
     fileSize: 8000000 // Compliant: 8MB
  }
});
```

body-parser module:

```
let jsonParser = bodyParser.json(); // Compliant, when the limit is not defined, the default value is set to 100kb
let urlencodedParser = bodyParser.urlencoded({ extended: false, limit: "2mb" }); // Compliant
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Owasp Cheat Sheet - Owasp Denial of Service Cheat Sheet
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-770 - Allocation of Resources Without Limits or Throttling
- CWE - CWE-400 - Uncontrolled Resource Consumption

Policies granting all privileges are security-sensitive

Clave: javascript:S6302

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

root_cause

A policy that grants all permissions may indicate an improper access control, which violates the principle of least privilege. Suppose an identity is granted full permissions to a resource even though it only requires read permission to work as expected. In this case, an unintentional overwriting of resources may occur and therefore result in loss of information.

how_to_fix

Recommended Secure Coding Practices

It’s recommended to apply the least privilege principle, i.e. by only granting the necessary permissions to identities. A good practice is to start with the very minimum set of permissions and to refine the policy over time. In order to fix overly permissive policies already deployed in production, a strategy could be to review the monitored activity in order to reduce the set of permissions to those most used.

Compliant Solution

A customer-managed policy that grants only the required permissions:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["iam:GetAccountSummary"],
    resources: ["arn:aws:iam:::user/*"],
})
```

See

- AWS Documentation - Grant least privilege
- Google Cloud Documentation - Understanding roles
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control

default

A policy that grants all permissions may indicate an improper access control, which violates the principle of least privilege. Suppose an identity is granted full permissions to a resource even though it only requires read permission to work as expected. In this case, an unintentional overwriting of resources may occur and therefore result in loss of information.

Ask Yourself Whether

Identities obtaining all the permissions:

- only require a subset of these permissions to perform the intended function.
- have monitored activity showing that only a subset of these permissions is actually used.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to apply the least privilege principle, i.e. by only granting the necessary permissions to identities. A good practice is to start with the very minimum set of permissions and to refine the policy over time. In order to fix overly permissive policies already deployed in production, a strategy could be to review the monitored activity in order to reduce the set of permissions to those most used.

Sensitive Code Example

A customer-managed policy that grants all permissions by using the wildcard (*) in the `Action` property:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["*"], // Sensitive
    resources: ["arn:aws:iam:::user/*"],
})
```

Compliant Solution

A customer-managed policy that grants only the required permissions:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["iam:GetAccountSummary"],
    resources: ["arn:aws:iam:::user/*"],
})
```

See

- AWS Documentation - Grant least privilege
- Google Cloud Documentation - Understanding roles
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-284 - Improper Access Control

assess_the_problem

Ask Yourself Whether

Identities obtaining all the permissions:

- only require a subset of these permissions to perform the intended function.
- have monitored activity showing that only a subset of these permissions is actually used.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

A customer-managed policy that grants all permissions by using the wildcard (*) in the `Action` property:

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ["*"], // Sensitive
    resources: ["arn:aws:iam:::user/*"],
})
```

Using unencrypted RDS DB resources is security-sensitive

Clave: javascript:S6303

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For `aws-cdk-lib.aws_rds.CfnDBCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBCluster(this, 'example', {
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.CfnDBInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBInstance(this, 'example', {
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';
import { aws_ec2 as ec2 } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

const cluster = new rds.DatabaseCluster(this, 'example', {
  engine: rds.DatabaseClusterEngine.auroraMysql({ version: rds.AuroraMysqlEngineVersion.VER_2_08_1 }),
  instanceProps: {
    vpcSubnets: {
      subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
    },
    vpc,
  },
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseClusterFromSnapshot`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseClusterFromSnapshot(this, 'example', {
  engine: rds.DatabaseClusterEngine.aurora({ version: rds.AuroraEngineVersion.VER_1_22_2 }),
  instanceProps: {
    vpc,
  },
  snapshotIdentifier: 'exampleSnapshot',
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseInstance(this, 'example', {
  engine: rds.DatabaseInstanceEngine.POSTGRES,
  vpc,
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstanceReadReplica`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const sourceInstance: rds.DatabaseInstance;

new rds.DatabaseInstanceReadReplica(this, 'example', {
  sourceDatabaseInstance: sourceInstance,
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.BURSTABLE2, ec2.InstanceSize.LARGE),
  vpc,
  storageEncrypted: false, // Sensitive
});
```

how_to_fix

Recommended Secure Coding Practices

It is recommended to enable encryption at rest on any RDS DB resource, regardless of the engine.
In any case, no further maintenance is required as encryption at rest is fully managed by AWS.

Compliant Solution

For `aws-cdk-lib.aws_rds.CfnDBCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBCluster(this, 'example', {
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.CfnDBInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBInstance(this, 'example', {
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.DatabaseCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

const cluster = new rds.DatabaseCluster(this, 'example', {
  engine: rds.DatabaseClusterEngine.auroraMysql({ version: rds.AuroraMysqlEngineVersion.VER_2_08_1 }),
  instanceProps: {
    vpcSubnets: {
      subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
    },
    vpc,
  },
  storageEncrypted: false, // Sensitive
});
```

For aws-cdk-lib.aws_rds.DatabaseClusterFromSnapshot:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseClusterFromSnapshot(this, 'example', {
  engine: rds.DatabaseClusterEngine.aurora({ version: rds.AuroraEngineVersion.VER_1_22_2 }),
  instanceProps: {
    vpc,
  },
  snapshotIdentifier: 'exampleSnapshot',
  storageEncrypted: true,
});
```

For aws-cdk-lib.aws_rds.DatabaseInstance:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseInstance(this, 'example', {
  engine: rds.DatabaseInstanceEngine.POSTGRES,
  vpc,
  storageEncrypted: true,
});
```

For aws-cdk-lib.aws_rds.DatabaseInstanceReadReplica:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const sourceInstance: rds.DatabaseInstance;

new rds.DatabaseInstanceReadReplica(this, 'example', {
  sourceDatabaseInstance: sourceInstance,
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.BURSTABLE2, ec2.InstanceSize.LARGE),
  vpc,
  storageEncrypted: true,
});
```

See

- AWS Documentation - Encrypting Amazon RDS resources
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

root_cause

Using unencrypted RDS DB resources exposes data to unauthorized access.

This includes database data, logs, automatic backups, read replicas, snapshots, and cluster metadata.

This situation can occur in a variety of scenarios, such as:

- A malicious insider working at the cloud provider gains physical access to the storage device.
- Unknown attackers penetrate the cloud provider’s logical infrastructure and systems.

After a successful intrusion, the underlying applications are exposed to:

- theft of intellectual property and/or personal data
- extortion
- denial of services and security bypasses via data corruption or deletion

AWS-managed encryption at rest reduces this risk with a simple switch.

default

Using unencrypted RDS DB resources exposes data to unauthorized access.

This includes database data, logs, automatic backups, read replicas, snapshots, and cluster metadata.

This situation can occur in a variety of scenarios, such as:

- A malicious insider working at the cloud provider gains physical access to the storage device.
- Unknown attackers penetrate the cloud provider’s logical infrastructure and systems.

After a successful intrusion, the underlying applications are exposed to:

- theft of intellectual property and/or personal data
- extortion
- denial of services and security bypasses via data corruption or deletion

AWS-managed encryption at rest reduces this risk with a simple switch.

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to enable encryption at rest on any RDS DB resource, regardless of the engine.

In any case, no further maintenance is required as encryption at rest is fully managed by AWS.

Sensitive Code Example

For `aws-cdk-lib.aws_rds.CfnDBCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBCluster(this, 'example', {
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.CfnDBInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBInstance(this, 'example', {
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';
import { aws_ec2 as ec2 } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

const cluster = new rds.DatabaseCluster(this, 'example', {
  engine: rds.DatabaseClusterEngine.auroraMysql({ version: rds.AuroraMysqlEngineVersion.VER_2_08_1 }),
  instanceProps: {
    vpcSubnets: {
      subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
    },
    vpc,
  },
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseClusterFromSnapshot`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseClusterFromSnapshot(this, 'example', {
  engine: rds.DatabaseClusterEngine.aurora({ version: rds.AuroraEngineVersion.VER_1_22_2 }),
  instanceProps: {
    vpc,
  },
  snapshotIdentifier: 'exampleSnapshot',
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseInstance(this, 'example', {
  engine: rds.DatabaseInstanceEngine.POSTGRES,
  vpc,
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstanceReadReplica`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const sourceInstance: rds.DatabaseInstance;
```

```
new rds.DatabaseInstanceReadReplica(this, 'example', {
  sourceDatabaseInstance: sourceInstance,
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.BURSTABLE2, ec2.InstanceSize.LARGE),
  vpc,
  storageEncrypted: false, // Sensitive
});
```

Compliant Solution

For `aws-cdk-lib.aws_rds.CfnDBCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBCluster(this, 'example', {
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.CfnDBInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

new rds.CfnDBInstance(this, 'example', {
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.DatabaseCluster`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

const cluster = new rds.DatabaseCluster(this, 'example', {
  engine: rds.DatabaseClusterEngine.auroraMysql({ version: rds.AuroraMysqlEngineVersion.VER_2_08_1 }),
  instanceProps: {
    vpcSubnets: {
      subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
    },
    vpc,
  },
  storageEncrypted: false, // Sensitive
});
```

For `aws-cdk-lib.aws_rds.DatabaseClusterFromSnapshot`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseClusterFromSnapshot(this, 'example', {
  engine: rds.DatabaseClusterEngine.aurora({ version: rds.AuroraEngineVersion.VER_1_22_2 }),
  instanceProps: {
    vpc,
  },
  snapshotIdentifier: 'exampleSnapshot',
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstance`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const vpc: ec2.Vpc;

new rds.DatabaseInstance(this, 'example', {
  engine: rds.DatabaseInstanceEngine.POSTGRES,
  vpc,
  storageEncrypted: true,
});
```

For `aws-cdk-lib.aws_rds.DatabaseInstanceReadReplica`:

```
import { aws_rds as rds } from 'aws-cdk-lib';

declare const sourceInstance: rds.DatabaseInstance;

new rds.DatabaseInstanceReadReplica(this, 'example', {
  sourceDatabaseInstance: sourceInstance,
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.BURSTABLE2, ec2.InstanceSize.LARGE),
  vpc,
  storageEncrypted: true,
});
```

See

- AWS Documentation - Encrypting Amazon RDS resources
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

Using unencrypted Elasticsearch domains is security-sensitive

Clave: javascript:S6308

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

default

Amazon OpenSearch Service is a managed service to host OpenSearch instances. It replaces Elasticsearch Service, which has been deprecated.

To harden domain (cluster) data in case of unauthorized access, OpenSearch provides data-at-rest encryption if the engine is OpenSearch (any version), or Elasticsearch with a version of 5.1 or above. Enabling encryption at rest will help protect:

- indices
- logs
- swap files
- data in the application directory
- automated snapshots

Thus, adversaries cannot access the data if they gain physical access to the storage medium.

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to encrypt OpenSearch domains that contain sensitive information.

OpenSearch handles encryption and decryption transparently, so no further modifications to the application are necessary.

Sensitive Code Example

For aws-cdk-lib.aws_opensearchservice.Domain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleDomain = new opensearchservice.Domain(this, 'ExampleDomain', {
  version: EngineVersion.OPENSEARCH_1_3,
}); // Sensitive, encryption must be explicitly enabled
```

For aws-cdk-lib.aws_opensearchservice.CfnDomain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleCfnDomain = new opensearchservice.CfnDomain(this, 'ExampleCfnDomain', {
  engineVersion: 'OpenSearch_1.3',
}); // Sensitive, encryption must be explicitly enabled
```

Compliant Solution

For aws-cdk-lib.aws_opensearchservice.Domain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleDomain = new opensearchservice.Domain(this, 'ExampleDomain', {
  version: EngineVersion.OPENSEARCH_1_3,
  encryptionAtRest: {
    enabled: true,
  },
});
```

For aws-cdk-lib.aws_opensearchservice.CfnDomain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleCfnDomain = new opensearchservice.CfnDomain(this, 'ExampleCfnDomain', {
  engineVersion: 'OpenSearch_1.3',
  encryptionAtRestOptions: {
    enabled: true,
  },
});
```

See

- AWS Documentation - Encryption of data at rest for Amazon OpenSearch Service
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

assess_the_problem

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For aws-cdk-lib.aws_opensearchservice.Domain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleDomain = new opensearchservice.Domain(this, 'ExampleDomain', {
  version: EngineVersion.OPENSEARCH_1_3,
}); // Sensitive, encryption must be explicitly enabled
```

For aws-cdk-lib.aws_opensearchservice.CfnDomain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleCfnDomain = new opensearchservice.CfnDomain(this, 'ExampleCfnDomain', {
  engineVersion: 'OpenSearch_1.3',
}); // Sensitive, encryption must be explicitly enabled
```

root_cause

Amazon OpenSearch Service is a managed service to host OpenSearch instances. It replaces Elasticsearch Service, which has been deprecated.

To harden domain (cluster) data in case of unauthorized access, OpenSearch provides data-at-rest encryption if the engine is OpenSearch (any version), or Elasticsearch with a version of 5.1 or above. Enabling encryption at rest will help protect:

- indices
- logs
- swap files
- data in the application directory
- automated snapshots

Thus, adversaries cannot access the data if they gain physical access to the storage medium.

how_to_fix

Recommended Secure Coding Practices

It is recommended to encrypt OpenSearch domains that contain sensitive information.

OpenSearch handles encryption and decryption transparently, so no further modifications to the application are necessary.

Compliant Solution

For aws-cdk-lib.aws_opensearchservice.Domain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleDomain = new opensearchservice.Domain(this, 'ExampleDomain', {
  version: EngineVersion.OPENSEARCH_1_3,
  encryptionAtRest: {
    enabled: true,
  },
});
```

For aws-cdk-lib.aws_opensearchservice.CfnDomain:

```
import { aws_opensearchservice as opensearchservice } from 'aws-cdk-lib';

const exampleCfnDomain = new opensearchservice.CfnDomain(this, 'ExampleCfnDomain', {
  engineVersion: 'OpenSearch_1.3',
  encryptionAtRestOptions: {
    enabled: true,
  },
});
```

See

- AWS Documentation - Encryption of data at rest for Amazon OpenSearch Service
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

AWS IAM policies should limit the scope of permissions given

Clave: javascript:S6317

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

AWS Identity and Access Management (IAM) is the service that defines access to AWS resources. One of the core components of IAM is the policy which, when attached to an identity or a resource, defines its permissions. Policies granting permission to an identity (a user, a group or a role) are called identity-based policies. They add the ability to an identity to perform a predefined set of actions on a list of resources.

For such policies, it is easy to define very broad permissions (by using wildcard `"*"` permissions for example.) This is especially true if it is not yet clear which permissions will be required for a specific workload or use case. However, it is important to limit the amount of permissions that are granted and the amount of resources to which these permissions are granted. Doing so ensures that there are no users or roles that have more permissions than they need.

If this is not done, it can potentially carry security risks in the case that an attacker gets access to one of these identities.

What is the potential impact?

AWS IAM policies that contain overly broad permissions can lead to privilege escalation by granting users more access than necessary. They may be able to perform actions beyond their intended scope.

Privilege escalation

When IAM policies are too permissive, they grant users more privileges than necessary, allowing them to perform actions that they should not be able to. This can be exploited by attackers to gain unauthorized access to sensitive resources and perform malicious activities.

For example, if an IAM policy grants a user unrestricted access to all S3 buckets in an AWS account, the user can potentially read, write, and delete any object within those buckets. If an attacker gains access to this user’s credentials, they can exploit this overly permissive policy to exfiltrate sensitive data, modify or delete critical files, or even launch further attacks within the AWS environment. This can have severe consequences, such as data breaches, service disruptions, or unauthorized access to other resources within the AWS account.

how_to_fix

In this example, the IAM policy allows an attacker to update the code of any Lambda function. An attacker can achieve privilege escalation by altering the code of a Lambda that executes with high privileges.

Noncompliant code example

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [new iam.PolicyStatement({
        effect: iam.Effect.ALLOW,
        actions: ["lambda:UpdateFunctionCode"],
        resources: ["*"], // Noncompliant
    })],
});
```

Compliant solution

The policy is narrowed such that only updates to the code of certain Lambda functions (without high privileges) are allowed.

```
import { aws_iam as iam } from 'aws-cdk-lib'

new iam.PolicyDocument({
    statements: [new iam.PolicyStatement({
        effect: iam.Effect.ALLOW,
        actions: ["lambda:UpdateFunctionCode"],
        resources: ["arn:aws:lambda:us-east-2:123456789012:function:my-function:1"],
    })],
});
```

How does this work?

Principle of least privilege

When creating IAM policies, it is important to adhere to the principle of least privilege. This means that any user or role should only be granted enough permissions to perform the tasks that they are supposed to, and *nothing else*.

To successfully implement this, it is easier to start from nothing and gradually build up all the needed permissions. When starting from a policy with overly broad permissions which is made stricter at a later time, it can be harder to ensure that there are no gaps that might be forgotten about. In this case, it might be useful to monitor the users or roles to verify which permissions are used.

resources

Documentation

- AWS Documentation - Policies and permissions in IAM: Grant least privilege

Articles & blog posts

- Rhino Security Labs - AWS IAM Privilege Escalation - Methods and Mitigation

Standards

- CWE - CWE-269 - Improper Privilege Management

introduction

Within IAM, identity-based policies grant permissions to users, groups, or roles, and enable specific actions to be performed on designated resources. When an identity policy inadvertently grants more privileges than intended, certain users or roles might be able to perform more actions than expected. This can lead to potential security risks, as it enables malicious users to escalate their privileges from a lower level to a higher level of access.

Using unencrypted SageMaker notebook instances is security-sensitive

Clave: javascript:S6319

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The instance contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For `aws-cdk-lib.aws-sagemaker.CfnNotebookInstance`

```
import { CfnNotebookInstance } from 'aws-cdk-lib/aws-sagemaker';

new CfnNotebookInstance(this, 'example', {
      instanceType: 'instanceType',
      roleArn: 'roleArn'
}); // Sensitive
```

default

Amazon SageMaker is a managed machine learning service in a hosted production-ready environment. To train machine learning models, SageMaker instances can process potentially sensitive data, such as personal information that should not be stored unencrypted. In the event that adversaries physically access the storage media, they cannot decrypt encrypted data.

Ask Yourself Whether

- The instance contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to encrypt SageMaker notebook instances that contain sensitive information. Encryption and decryption are handled transparently by SageMaker, so no further modifications to the application are necessary.

Sensitive Code Example

For `aws-cdk-lib.aws-sagemaker.CfnNotebookInstance`

```
import { CfnNotebookInstance } from 'aws-cdk-lib/aws-sagemaker';

new CfnNotebookInstance(this, 'example', {
      instanceType: 'instanceType',
      roleArn: 'roleArn'
}); // Sensitive
```

Compliant Solution

For `aws-cdk-lib.aws-sagemaker.CfnNotebookInstance`

```
import { CfnNotebookInstance } from 'aws-cdk-lib/aws-sagemaker';

const encryptionKey = new Key(this, 'example', {
    enableKeyRotation: true,
});
new CfnNotebookInstance(this, 'example', {
    instanceType: 'instanceType',
    roleArn: 'roleArn',
    kmsKeyId: encryptionKey.keyId
});
```

See

- Protect Data at Rest Using Encryption
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

how_to_fix

Recommended Secure Coding Practices

It’s recommended to encrypt SageMaker notebook instances that contain sensitive information. Encryption and decryption are handled transparently by SageMaker, so no further modifications to the application are necessary.

Compliant Solution

For `aws-cdk-lib.aws-sagemaker.CfnNotebookInstance`

```
import { CfnNotebookInstance } from 'aws-cdk-lib/aws-sagemaker';

const encryptionKey = new Key(this, 'example', {
    enableKeyRotation: true,
});
new CfnNotebookInstance(this, 'example', {
    instanceType: 'instanceType',
    roleArn: 'roleArn',
    kmsKeyId: encryptionKey.keyId
});
```

See

- Protect Data at Rest Using Encryption
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

root_cause

Amazon SageMaker is a managed machine learning service in a hosted production-ready environment. To train machine learning models, SageMaker instances can process potentially sensitive data, such as personal information that should not be stored unencrypted. In the event that adversaries physically access the storage media, they cannot decrypt encrypted data.

React "render" functions should return a value

Clave: javascript:S6435

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Component
- React Documentation - `render()`

root_cause

In React, the `render` function is a required method in a class component that defines what will be rendered to the user interface (UI). It is responsible for returning a value, typically a JSX (JavaScript XML) expression, that describes the structure and appearance of the component’s UI.

When writing the `render` function in a component, it is easy to forget to return the JSX content, which means the component will render nothing. Thus having a `render` function without a single `return` statement is usually a mistake.

```
const React = require('react');
class MyComponent extends React.Component {
  render() {
    <div>Contents</div>; // Noncompliant: The render function returns nothing
  }
}
```

Make sure that the `render` function returns the JSX expression describing the structure and appearance of the component.

```
const React = require('react');
class MyComponent extends React.Component {
  render() {
    return <div>Contents</div>;
  }
}
```

If it’s required that the component renders nothing, the `render` function should explicitly return `null`.

```
const React = require('react');
class MyComponent extends React.Component {
  render() {
    return null;
  }
}
```

Comments inside JSX expressions should be enclosed in curly braces

Clave: javascript:S6438

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - JavaScript in JSX with Curly Braces

root_cause

JSX lets you write HTML-like markup inside a JavaScript file, commonly used in React.

Adding comments inside JSX might be tricky as JSX code is neither a plain HTML nor JavaScript.

HTML comments (`<!-- comment here -->`) are not valid syntax in JSX.

JavaScript-style comments, single or multiline, will create an additional text node in the browser, which is probably not expected.

```
<div>
  // Noncompliant: text inside node
</div>
```

To avoid that, use JavaScript multiline comments enclosed in curly braces. Single-line comments can also be used, but avoid having the ending bracket in the same line.

```
<div>
  {
    /*
      multi-line
      comment
    */
  }
  {
    // single-line comment
  }
  { /* short form comment */ }
</div>
```

Note that JavaScript comments around attributes are also allowed (`<div /* comment */>`).

If the additional text node is intentional, prefer using a JavaScript string literal containing that comment.

```
<div>
  { '// text inside node' }
</div>
```

React components should not render non-boolean condition values

Clave: javascript:S6439

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Logical AND (&&) operator is sometimes used to conditionally render in React (aka short-circuit evaluation). For example, `myCondition && <MyElement />` will return `<MyElement />` if `myCondition` is `true` and `false` otherwise.

React considers `false` as a 'hole' in the JSX tree, just like `null` or `undefined`, and doesn’t render anything in its place. But if the condition has a `falsy` non-boolean value (e.g. `0`), that value will leak into the rendered result.

This rule will report when the condition has type `number` or `bigint`.

In the case of React Native, the type `string` will also raise an error, as your render method will crash if you render `0`, `''`, or `NaN`.

```
function Profile(props) {
  return <div>
    <h1>{ props.username }</h1>
    { props.orders && <Orders /> } { /* Noncompliant: 0 will be rendered if no orders available */ }
  </div>;
}
```

Instead, make the left-hand side a boolean to avoid accidental renderings.

```
function Profile(props) {
  return <div>
    <h1>{ props.username }</h1>
    { props.orders > 0 && <Orders /> }
  </div>;
}
```

Another alternative to achieve conditional rendering is using the ternary operator (`myCondition ? <MyElement /> : null`), which is less error-prone in this case as both return values are explicit.

```
function Profile(props) {
  return <div>
    <h1>{ props.username }</h1>
    { props.orders ? <Orders /> : null }
  </div>;
}
```

resources

Documentation

- React Documentation - Conditional Rendering

Optional chaining should not be used if returning "undefined" throws an error

Clave: javascript:S6523

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

In order to prevent runtime errors, you should provide fallbacks for when the optional chaining operator short-circuits to `undefined`.

- Nullish coalescing operator `??`

```
(event?.callback ?? defaultCallback)();
```

- Logical OR operator `||`

```
(event?.callback || defaultCallback)();
```

- Ternary operator `condition ? exprIfTrue : exprIfFalse`

```
(event?.callback ? event?.callback : defaultCallback)();
```

root_cause

The optional chaining operator `?.` allows to access a deeply nested property, returning `undefined` if the property or any intermediate object is `undefined`.

This usually means that the expression is expected to evaluate as `undefined` in some cases. Therefore, using the optional chaining operator in a context where returning `undefined` throws an error can lead to runtime exceptions.

```
(event?.callback)(); // Noncompliant: when 'event' does not have 'callback' property TypeError is thrown
const { code } = event?.error; // Noncompliant: when 'event' does not have 'error' property TypeError is thrown
func(...event?.values); // Noncompliant: when 'event' does not have 'values' property TypeError is thrown
```

Since optional chaining represents multiple execution branches, having an error thrown in such a context can be hard to debug.

resources

Documentation

- MDN web docs - Optional chaining (?.)
- MDN web docs - Nullish coalescing operator (??)
- MDN web docs - Logical OR (||)
- MDN web docs - Conditional (ternary) operator

Promises should not be misused

Clave: javascript:S6544

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Promises need to be resolved or awaited to return the expected value, otherwise, they return the promise object.

Unresolved promises:

Forgetting to await a promise is a frequent mistake. There are places where the use of a promise object is confusing or unclear because the developer forgot to resolve it.

This rule forbids returning promises where another type is expected such as in:

- conditionals
- void returns
- spread operators

What is the potential impact?

Using a promise instead of its resolved value can have unexpected results leading to bugs.

- In conditionals, it will always return a truthy value.
- In places where the expected type is void, returning a promise is often a mistake.

- Using the spread operator on a promise will raise an exception.

The executor function of a promise can also be an async function. However, this usually denotes a mistake:

- If an async executor function throws an error, the error won’t cause the created promise to reject and will be lost. Therefore, this could make it difficult to debug and handle runtime errors.
- If a promise executor function is using `await`, this means that it’s not necessary to use the `Promise` constructor, or the scope of the Promise constructor can be reduced.

Exceptions

This rule can be ignored for promises that you know will always resolve like timers.

```
await new Promise(resolve => time.setTimeout(1000));
```

resources

Documentation

- MDN web docs - Promise
- MDN web docs - Using promises
- MDN web docs - Async function
- MDN web docs - IIFE

how_to_fix

If you mistakenly treated a promise as its resolved value, you can ensure it is properly resolved by using await or resolve on the promise. In some cases, you may need to use an "immediately invoked function expression" (IIFE):

```
(async function foo() {
  const result = await bar();
  // work with result
})();
```

Noncompliant code example

```
const promise = new Promise((resolve, reject) => {
  // ...
  resolve(false)
});
if (promise) {
  // ...
}
```

Compliant solution

```
const promise = new Promise((resolve, reject) => {
  // ...
  resolve(false)
});
if (await promise) {
  // ...
}
```

Noncompliant code example

```
const p = new Promise(async (resolve, reject) => {
  doSomething('Hey, there!', function(error, result) {
    if (error) {
      reject(error);
      return;
    }
    await saveResult(result)
    resolve(result);
  });
});

await p;
```

Compliant solution

```
const p = new Promise((resolve, reject) => {
  doSomething('Hey, there!', function(error, result) {
    if (error) {
      reject(error);
      return;
    }
    resolve(result);
  });
});

const result = await p;
await saveResult(result);
```

Noncompliant code example

```
apiCalls.forEach(async (apiCall) => {
  await apiCall.send();
});
```

Compliant solution

```
for (const apiCall of apiCalls) {
  await apiCall.send();
}
```

How does this work?

In JavaScript, a promise is a mechanism to perform tasks asynchronously. To this end, the language provides the `Promise` object which represents the eventual completion or failure of an asynchronous operation and its resulting value. A promise can be created with the `Promise` constructor accepting an executor function as an argument, which has `resolve` and `reject` parameters that are invoked when the promise completes or fails.

The logic of the promise is executed when it is called, however, its result is obtained only when the promise is resolved or awaited.

Objects and classes converted or coerced to strings should define a "toString()" method

Clave: javascript:S6551

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When using an object in a string context, a developer wants to get the string representation of the state of an object, so obtaining `[object Object]` is probably not the intended behaviour and might even denote a bug.

resources

Documentation

- MDN web docs - `Object.prototype.toString()`

introduction

When calling `toString()` or coercing into a string an object that doesn’t implement its own `toString` method, it returns `[object Object]` which is often not what was intended.

how_to_fix

You can simply define a `toString()` method for the object or class.

Noncompliant code example

```
class Foo {};
const foo = new Foo();

foo + ''; // Noncompliant - evaluates to "[object Object]"
`Foo: ${foo}`; // Noncompliant - evaluates to "Foo: [object Object]"
foo.toString(); // Noncompliant - evaluates to "[object Object]"
```

Compliant solution

```
class Foo {
  toString() {
    return 'Foo';
  }
}
const foo = new Foo();

foo + '';
`Foo: ${foo}`;
foo.toString();
```

Noncompliant code example

```
const foo = {};
foo + ''; // Noncompliant - evaluates to "[object Object]"
`Foo: ${foo}`; // Noncompliant - evaluates to "Foo: [object Object]"
foo.toString(); // Noncompliant - evaluates to "[object Object]"
```

Compliant solution

```
const foo = {
  toString: () => {
    return 'Foo';
  }
}
foo + '';
`Foo: ${foo}`;
foo.toString();
```

Ends of strings should be checked with "startsWith()" and "endsWith()"

Clave: javascript:S6557

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

One should use `String#startsWith` to check the start of a string and `String#endsWith` to check the end.

Noncompliant code example

```
const str = 'abc';

str[0] === 'a';
str.charAt(0) === 'a';
str.indexOf('abc') === 0;
str.slice(0, 3) === 'abc';
str.substring(0, 3) === 'abc';
str.match(/^abc/) != null;
/^abc/.test(str);
```

Compliant solution

```
str.startsWith('abc');
```

Noncompliant code example

```
const str = 'abc';

str[str.length - 1] === 'c';
str.charAt(str.length - 1) === 'c';
str.lastIndexOf('abc') === str.length - 3;
str.slice(-3) === 'abc';
str.substring(str.length - 3) === 'abc';
str.match(/abc$/) != null;
/abc$/.test(str);
```

Compliant solution

```
str.endsWith('abc');
```

root_cause

When writing code, it is quite common to test patterns against string ends. For a long time, JavaScript did not provide proper support for this use case. As a result, developers have been relying on various programming subtleties to check the start or end of a string. Examples are getting the index of a substring, slicing the beginning of a string, extracting a substring from the head, matching a regular expression beginning or ending with a pattern, and so on.

While these approaches are all technically valid, they look more like hacking than anything else, blur the developer’s intent, but more importantly affect code readability.

Since ES2015, JavaScript provides `String#startsWith` and `String#endsWith`, which are the preferred ways to test patterns against string ends.

resources

Documentation

- MDN web docs - String#startsWith
- MDN web docs - String#endsWith

Ternary operator should not be used instead of simpler alternatives

Clave: javascript:S6644

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Ternary operator
- MDN web docs - Logical NOT (!)
- MDN web docs - Logical OR (||)

root_cause

Ternary operator should not be used to select between two boolean values, or instead of a logical `OR` operation. Ternary expressions are often difficult to read, so if a simpler syntax exists, it should be used instead of a ternary expression. This happens when

- the expression returns two boolean values

```
let isGood = value > 0 ? true : false; // Non-compliant, replace with value > 0
let isBad = value > 0 ? false : true; // Non-compliant, replace with !(value > 0)
```

- the same value is used for both the conditional test and the consequent

```
let a = x ? x : y;  // Non-compliant, replace with x || y
```

Variables should not be initialized to undefined

Clave: javascript:S6645

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Initializing a variable to `undefined` is unnecessary and should be avoided. A variable will automatically be set to `undefined` if you declare it without initialization, so the initialization code is redundant in this case.

```
var foo = undefined; // Noncompliant: replace with var foo;
let bar = undefined; // Noncompliant: replace with let foo;
```

resources

Documentation

- MDN web docs - `undefined`
- MDN web docs - `let`
- MDN web docs - `var`
- MDN web docs - hoisting

Unnecessary constructors should be removed

Clave: javascript:S6647

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

If the class declaration does not include a constructor, one is automatically created, so there is no need to provide an empty constructor, or one that just delegates to the parent class.

```
class Foo {
    constructor() {}  // Noncompliant, empty
}

class Bar extends Foo {
    constructor(params) { // Noncompliant: just delegates to the parent
        super(params);
    }
}
```

Instead, you can safely remove the empty constructor without affecting the functionality.

```
class Foo {}

class Bar extends Foo {}
```

resources

Documentation

- MDN web docs - constructor

If statements should not be the only statement in else blocks

Clave: javascript:S6660

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `if...else`

root_cause

When `if` is the only statement in the `else` block, it is better to use `else if` because it simplifies the code and makes it more readable.

When using nested `if` statements, it can be difficult to keep track of the logic and understand the flow of the code. Using `else if` makes the code more concise and easier to follow.

```
if (condition1) {
    // ...
} else {
    if (condition2) {  // Noncompliant: 'if' statement is the only statement in the 'else' block
        // ...
    }
}


if (condition3) {
    // ...
} else {
    if (condition4) { // Noncompliant: 'if' statement is the only statement in the 'else' block
        // ...
    } else {
        // ...
    }
}
```

Fix your code by using `else if` if the nested `if` is the only statement in the `else` block.

```
if (condition1) {
    // ...
} else if (condition2) {
    // ...
}


if (condition3) {
    // ...
} else if (condition4) {
    // ...
} else {
    // ...
}
```

Object spread syntax should be used instead of "Object.assign"

Clave: javascript:S6661

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Spread in object literals
- MDN web docs - `Object.assign()`

root_cause

When merging objects or copying properties from one object to another, use the object spread syntax instead of `Object.assign()`. The object spread syntax was introduced in ES2018 and allows shallow-cloning or merging of objects with a more concise and readable syntax.

The `Object.assign()` also allows to mutate an object, which is not possible with the spread syntax, so the rule only applies to cases where the first argument of the `Object.assign()` is an object literal.

The object spread syntax improves clarity when you’re modifying an object, as demonstrated in this example: `foo = { bar: 42, …baz }`. Additionally, it provides a more concise way to perform a shallow clone. Instead of using `foo = Object.assign({}, bar)`, you can simply write `foo = { …bar }`.

```
const a = Object.assign({}, foo); // Noncompliant: Use spread syntax to clone or merge objects
const b = Object.assign({}, foo, bar); // Noncompliant: Use spread syntax to clone or merge objects
const c = Object.assign({foo: 123}, bar); // Noncompliant: Use spread syntax to clone or merge objects
const d = Object.assign({}); // Noncompliant: Use spread syntax to clone or merge objects
```

To fix the code replace `Object.assign()` with a spread syntax.

```
const a = {...foo};
const b = {...foo, ...bar};
const c = {foo: 123, ...bar};
const d = {};
```

Spread syntax should be used instead of "apply()"

Clave: javascript:S6666

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - spread syntax
- MDN web docs - `apply()`

root_cause

The spread operator is a more concise and more readable way to pass arguments to a function that takes a variable number of arguments (variadic function). Prior to ES2015, the only way to call such functions with a variable number of arguments was to use the `.apply()` method.

```
foo.apply(undefined, args); // Noncompliant: use spread syntax instead of .apply()
foo.apply(null, args); // Noncompliant: use spread syntax instead of .apply()
obj.foo.apply(obj, args); // Noncompliant: use spread syntax instead of .apply()
```

Using `.apply()` is no longer necessary in such cases - replace it with a spread operator applied to the array of arguments.

```
foo( ...args);
foo( ...args);
obj.foo( ...args);
```

Literals should not be used for promise rejection

Clave: javascript:S6671

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Promise
- MDN web docs - Error

Related rules

- S3696 - Literals should not be thrown

root_cause

The use of literals (primitive values such as strings, numbers, booleans, etc.) for promise rejection is generally discouraged. While it is syntactically valid to provide literals as a rejected promise value, it is considered best practice to use instances of the Error class or its subclasses instead.

Using an instance of the Error class allows you to provide more meaningful information about the error. The Error class and its subclasses provide properties such as message and stack that can be used to convey useful details about the error, such as a description of the problem, the context in which it occurred, or a stack trace for debugging.

```
new Promise(function(resolve, reject) {
  reject(); // Noncompliant: use Error object to provide rejection reason
});

new Promise(function(resolve, reject) {
  reject('Something went wrong'); // Noncompliant: use Error object instead of literal
});
```

To fix your code provide an instanse of the Error class to the promise reject function.

```
new Promise(function(resolve, reject) {
  reject(new Error('Network timeout'));
});

new Promise(function(resolve, reject) {
  reject(new Error('Something went wrong'));
});
```

Calls to ".call()" and ".apply()" methods should not be redundant

Clave: javascript:S6676

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Functions
- MDN web docs - Function.prototype.call()
- MDN web docs - Function.prototype.apply()
- MDN web docs - this
- MDN web docs - globalThis

root_cause

If the function call can be written in a normal way, then calling that function with `.call()` or `.apply()` methods is redundant and can be removed without affecting the behavior of the code.

The `.call()` and `.apply()` methods are traditionally used to explicitely set the value of `this` keyword when executing a function or an object method. When calling a method of an object the value of `this` by default will be the reference to that object. But if you call a function or a method using `.call()` and `.apply()` you can set the value of `this` to any object, whatever you put into the first argument.

```
let obj = {
    checkThis() {
        this === obj; // true, if called the normal way: obj.checkThis()
    }
};

let otherObject = {};

obj.checkThis.call(otherObject); // this === otherObject, if called this way
```

There is also a special case when your code is not in strict mode and the first argument to the `.call()` and `.apply()` methods is `null` or `undefined`. In this case the value of `this` is substituted with the `globalThis` object, usually `window` when in browser context, and it make such call equivalent to just calling the function the normal way.

So if you are calling a function using `.call()` or `.apply()` methods and the first argument is `null` or `undefined`, or you are calling an object method and the first argument is the object itself, then `.call()` or `.apply()` methods become redundant, and should be removed to make the code more simple and easier to understand.

```
foo.call(null, 1, 2); // Noncompliant: .call() is redundant
obj.foo.call(obj, arg1, arg2); // Noncompliant: .call() is redundant
bar.apply(undefined, [x, y, z]); // Noncompliant: .apply() is redundant
```

To fix your code remove redundant `.call()` or `.apply()` methods.

```
foo(1, 2);
obj.foo(arg1, arg2);
bar(x, y, z);
```

"Number.isNaN()" should be used to check for "NaN" value

Clave: javascript:S6679

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `NaN`
- MDN web docs - `Number.isNaN()`
- MDN web docs - `isNaN()`
- MDN web docs - Type conversion

Related rules

- S1764 - Identical expressions should not be used on both sides of a binary operator
- S2688 - "NaN" should not be used in comparisons

root_cause

Comparing the value to itself may be either a refactoring error or an outdated way of checking if the value is of a numeric data type but not a valid number. This special numeric value is represented by `NaN` global property, where `NaN` stands for "Not-a-Number". `NaN` is returned as a result when an arithmetic operation or mathematical function is performed, and the result is undefined or unrepresentable as a valid number.

Detecting whether a value is `NaN` in JavaScript was previously problematic because of the way `NaN` behaves in comparison operations. `NaN` is not equal to any value, including itself, so comparing the value to `NaN` will always return false. In other words, if a value is not equal to itself, it can only be `NaN`.

This method of detecting `NaN` can be confusing and should be avoided. ES6 introduced a special function `Number.isNaN()` which only returns `true` if the argument is `NaN` value. For clarity and consistency this function should be used to detect `NaN` instead of all other methods.

```
if (value !== value){ // Noncompliant: use Number.isNaN()
    processNaN(value);
}
```

To fix your code replace self-comparison with `Number.isNaN()` function.

```
if (Number.isNaN(value)){
    processNaN(value);
}
```

Do not confuse `Number.isNaN()` with the legacy global `isNaN()` function. While they serve a similar purpose, the behavior is very different - the global `isNaN()` tries to convert its argument into a number before checking if it is `NaN`. If the argument cannot be converted into a number, `isNaN()` will return true, which may not be the desired behavior in all cases.

```
isNaN('some text');          // true: Number('some text') returns NaN
Number.isNaN('some text');   // false: 'some text' is not NaN
```

You should use the `Number.isNaN()` method over `isNaN()` to perform a strict check for `NaN` without any type conversion.

If the intention was not to detect whether a value was NaN, fix the issue by not comparing the variable against itself.

```
if (value !== anotherValue){
    // ...
}
```

"shouldComponentUpdate" should not be defined when extending "React.PureComponent"

Clave: javascript:S6763

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

In React, a `PureComponent` is a class component that is optimized for performance by implementing a shallow comparison of props and state. It is a subclass of the regular React `Component` class and provides a default implementation of the `shouldComponentUpdate` method.

The `shouldComponentUpdate` method is responsible for determining whether a component should re-render or not. By default, it returns true and triggers a re-render every time the component receives new props or state. However, `PureComponent` overrides this method and performs a shallow comparison of the current and next props and state. If there are no changes, it prevents unnecessary re-renders by returning false.

Therefore, defining a `shouldComponentUpdate` method while extending `PureComponent` is redundant and should be avoided. By not defining `shouldComponentUpdate`, you allow `React.PureComponent` to handle the optimization for you.

```
class MyComponent extends React.PureComponent { // Noncompliant
  shouldComponentUpdate() {
    // does something
  }

  render() {
    return <div>Hello!</div>
  }
}
```

You should either remove the redundant method `shouldComponentUpdate` or turn your component into a regular one by extending `Component`.

```
class MyComponent extends React.Component {
  shouldComponentUpdate() {
    // does something
  }

  render() {
    return <div>Hello!</div>
  }
}
```

resources

Documentation

- React Documentation - shouldComponentUpdate
- React Documentation - PureComponent

JSX special characters should be escaped

Clave: javascript:S6766

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

If JSX special characters (`>`, `}`) appear unescaped in the element body, this may be either because you simply forgot to escape them or because there is a problem with the JSX tag or expression (for example, misplaced or duplicate closing `>` or `}` brackets).

```
<MyComponent
  name="abc"
  foo="bar">
  x="y"> {/* Noncompliant: closing > should only be on this line */}
  Body Text
</MyComponent>
```

To fix the issue, check the structure of your JSX tag or expression - are the closing brackets correct and in the right place? If the special character is there on purpose - you need to change it to the appropriate HTML entity.

- replace `>` with `&gt;`
- replace `}` with `&#125;`

```
<MyComponent
  name="abc"
  foo="bar"
  x="y">
  Body Text
</MyComponent>
```

The characters < and { should also be escaped, but they are not checked by this rule because it is a syntax error to include those tokens inside of a tag.

resources

Documentation

- React Documentation - Writing markup with JSX
- MDN web docs - Entity codes

Unused React typed props should be removed

Clave: javascript:S6767

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Passing Props to a Component
- React Documentation - static propTypes
- React Documentation - TypeScript with React Components

how_to_fix

Noncompliant code example

```
import React from 'react';

type Props = {
  name: string;
}

class Hello extends React.Component<Props> {
  render() {
    return <div>Hello</div>;
  }
}
```

Compliant solution

```
import React from 'react';

type Props = {
  name: string;
};

class Hello extends React.Component<Props> {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

root_cause

Leaving unused props in a React component can make the code harder to understand and maintain. Other developers may wonder why certain props are passed to a component if they are not used. Unused props can also increase the size of the component’s memory footprint and impact performance. This is especially true if the unused props are large objects or arrays. Furthermore, if a prop is unused, it may indicate that the developer did not complete the implementation as he intended initially or made a mistake while writing the component.

To avoid these issues, you should remove any unused props from React components. This helps keep the codebase clean, improves performance, and enhances code readability.

how_to_fix

Noncompliant code example

```
import PropTypes from 'prop-types';
import React from 'react';

class Hello extends React.Component {
  render() {
    return (
      <h1>Hello</h1>
    );
  }
}

Hello.propTypes = {
  name: PropTypes.string
};
```

Compliant solution

```
import PropTypes from 'prop-types';
import React from 'react';

class Hello extends React.Component {
  render() {
    return (
      <h1>Hello {this.props.name}</h1>
    );
  }
}

Hello.propTypes = {
  name: PropTypes.string
};
```

String references should not be used

Clave: javascript:S6790

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - `ref` callback function
- React Documentation - Refs and the DOM
- React Documentation - Manipulating the DOM with Refs
- React Documentation - `useRef`
- React Documentation - `createRef`

root_cause

React Refs provide a way to access DOM nodes or React elements created in the render method.

Older React versions allowed the ref attribute to be a string, like `"textInput"`, later accessible as `this.refs.textInput`. This is considered legacy code due to multiple reasons:

- String `refs` make React slower as they force React to keep track of what component is currently executing.
- String `refs` are not composable: if a library puts a ref on the passed child, the user can’t put another ref on it.
- The owner of a string `ref` is determined by the currently executing component.

```
const Hello = createReactClass({
  componentDidMount() {
    const component = this.refs.hello; // Noncompliant
    // ...
  },
  render() {
    return <div ref="hello">Hello, world.</div>;
  }
});
```

Instead, reference callbacks should be used. These do not have the limitations mentioned above. When the DOM node is added to the screen, React will call the `ref` callback with the DOM node as the argument. When that DOM node is removed, React will call your `ref` callback with `null`. One should return `undefined` from the `ref` callback.

```
const Hello = createReactClass({
  componentDidMount() {
    const component = this.hello;
    // ...
  },
  render() {
    return <div ref={(c) => { this.hello = c; }}>Hello, world.</div>;
  }
});
```

React legacy lifecycle methods should not be used

Clave: javascript:S6791

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Instead of `componentWillUpdate`, use `getSnapshotBeforeUpdate` together with `componentDidUpdate`. The `getSnapshotBeforeUpdate` lifecycle is called right before mutations are made. The return value for this lifecycle will be passed as the third parameter to `componentDidUpdate`.

Instead of `componentWillReceiveProps`, Use `getDerivedStateFromProps` together with `componentDidUpdate`. The `getDerivedStateFromProps` lifecycle is invoked after a component is instantiated as well as before it is re-rendered. It can return an object to update state, or null to indicate that the new props do not require any state updates.

As for `componentWillMount`, React will call it immediately after the constructor. It only exists for historical reasons and should not be used. Instead, use one of the alternatives:

- To initialize state, declare `state` as a class field or set `this.state` inside the `constructor`.
- If you need to run a side effect or set up a subscription, move that logic to `componentDidMount` instead.

Noncompliant code example

```
class myComponent extends React.Component {
  constructor(props) {
      super(props);
  }

  componentWillMount() { // Noncompliant: "componentWillMount" is deprecated
      if (localStorage.getItem("token")) {
          this.setState({logged_in: true});
      }
  }
  // ...
}
```

Compliant solution

```
class myComponent extends React.Component {
  constructor(props) {
      super(props);

      if (localStorage.getItem("token")) {
          this.setState({logged_in: true});
      }
  }
  // ...
}
```

resources

Documentation

- React Documentation - `UNSAFE_componentWillMount`
- React Documentation - `UNSAFE_componentWillReceiveProps`
- React Documentation - `UNSAFE_componentWillUpdate`
- React Documentation - Migrating from Legacy Lifecycles

root_cause

React works in two phases: render and commit.

- The render phase determines what changes need to be made to e.g. the DOM. During this phase, React calls render and then compares the result to the previous render.
- The commit phase is when React applies any changes. (In the case of React DOM, this is when React inserts, updates, and removes DOM nodes.) React also calls lifecycles like `componentDidMount` and `componentDidUpdate` during this phase.

Render phase lifecycles include among others, the following lifecycle methods:

- `componentWillMount` (or its alias `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or its alias `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or its alias `UNSAFE_componentWillUpdate`)

These are considered unsafe and also happen to be the lifecycles that cause the most confusion within the React community and tend to encourage unsafe coding practices.

```
class Foo extends React.Component {
  UNSAFE_componentWillMount() {}         // Noncompliant
  UNSAFE_componentWillReceiveProps() {}  // Noncompliant
  UNSAFE_componentWillUpdate() {}        // Noncompliant
}
```

ARIA properties in DOM elements should have valid values

Clave: javascript:S6793

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

ARIA (Accessible Rich Internet Applications) attributes are used to enhance the accessibility of web content and web applications. These attributes provide additional information about an element’s role, state, properties, and values to assistive technologies like screen readers.

This rule checks that the values of ARIA attributes "aria-*" in DOM elements are valid.

How to fix

Check that each element with a defined ARIA attribute has a valid value.

```
<span aria-hidden="ok">foo</span>
```

To fix the code use a valid value for the aria-* attribute.

```
<span aria-hidden="true">foo</span>
```

resources

Documentation

- MDN web docs - Using ARIA: Roles, states, and properties
- MDN web docs - ARIA states and properties (Reference)

Standards

- W3C - Accessible Rich Internet Applications (WAI-ARIA) 1.2

Function and method names should comply with a naming convention

Clave: javascript:S100

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Shared naming conventions allow teams to collaborate efficiently.

This rule raises an issue when a function or a method name does not match a provided regular expression.

For example, with the default regular expression `^[a-z][a-zA-Z0-9]*$`, the function:

```
function DoSomething(){...}  // Noncompliant
```

should be renamed to

```
function doSomething(){...}
```

Exceptions

This rule ignores React Functional Components, JavaScript functions named with a capital letter and returning a React element (JSX syntax).

```
function Welcome() { // Compliant by exception
  const greeting = 'Hello, World!';

  // ...

  return (
    <div className="Welcome">
      <p>{greeting}</p>
    </div>
  );
}
```

Class names should comply with a naming convention

Clave: javascript:S101

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Shared naming conventions allow teams to collaborate efficiently.

This rule raises an issue when a class name (or an interface for TypeScript) does not match a provided regular expression.

For example, with the default provided regular expression `^[A-Z][a-zA-Z0-9]*$`, the class:

```
class my_class {...} // Noncompliant
```

should be renamed to

```
class MyClass {...}
```

Mouse events should have corresponding keyboard events

Clave: javascript:S1082

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Add at least one of the following event handlers `onKeyUp`, `onKeyDown`, `onKeyPress` to the element when using `onClick` event handler. Add corresponding event handlers `onfocus`/`onblur` to the element when using `onmouseover`/`onmouseout` event handlers.

Noncompliant code example

```
<div onClick={() => {}} />
```

```
<div onMouseOver={ () => {}} }  />
```

Compliant solution

```
<div onClick={() => {}} onKeyDown={this.handleKeyDown} />
```

```
<div onMouseOver={ () => {} } onFocus={ () => {} } />
```

Exceptions

This does not apply for interactive or hidden elements, eg. when using `aria-hidden="true"` attribute.

root_cause

Offering the same experience with the mouse and the keyboard allow users to pick their preferred devices.

Additionally, users of assistive technology will also be able to browse the site even if they cannot use the mouse.

This rules detects the following issues:

- when `onClick` is not accompanied by at least one of the following: `onKeyUp`, `onKeyDown`, `onKeyPress`.

- when `onmouseover`/`onmouseout` are not paired by `onfocus`/`onblur`.

resources

- SCR2 - Using redundant keyboard and mouse event handlers
- G90 - Providing keyboard-triggered event handlers

String literals should not be duplicated

Clave: javascript:S1192

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Use constants to replace the duplicated string literals. Constants can be referenced from many places, but only need to be updated in a single place.

Noncompliant code example

With the default threshold of 3:

```
function run() {
    prepare("action_to_launch");  // Noncompliant - "action_to_launch" is duplicated 3 times
    execute("action_to_launch");
    release("action_to_launch");
}

function printInQuotes(a, b) {
  console.log("'" + a + "'" + b + "'");               // Compliant - literal "'" has less than 10 characters and is excluded
}
```

Compliant solution

```
var ACTION_1 = "action_to_launch";

function run() {
  prepare(ACTION_1);                               // Compliant
  execute(ACTION_1);
  release(ACTION_1);
}
```

root_cause

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.

Exceptions

To prevent generating some false-positives, literals having less than 10 characters are excluded as well as literals matching `/^\w*$/`. String literals inside import/export statements and JSX attributes are also ignored. The same goes for statement-like string literals, e.g. `'use strict';`.

Classes should not be empty

Clave: javascript:S2094

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Using an empty class serves no purpose and can hinder the readability of the code.

```
class Foo {
  static bar() {
    // ...
  }
}
```

how_to_fix

You can export the functions that you wish to make available.

```
export function bar()  {
  // ...
}
```

Noncompliant code example

```
class Foo { // Noncompliant
  static bar() {
    // ...
  }
}
```

Compliant solution

```
export function bar() {
  // ...
}
```

Noncompliant code example

```
class DoAndLog { // Noncompliant
  constructor () {
    console.log('I\'m done!');
  }
}
```

Compliant solution

```
function doAndLog() {
  console.log('I\'m done!');
}
```

introduction

There is no good excuse for an empty class. If it’s being used simply as a common extension point, it should be replaced with an `interface`. If it was stubbed in as a placeholder for future development it should be fleshed-out. In any other case, it should be eliminated.

Additionally, one shouldn’t use a class to define exclusively static methods. Instead one can use a module, or better, export each function separately.

Collection contents should be used

Clave: javascript:S4030

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A collection is a data structure that holds multiple values, such as an array or a map. If a collection is declared and populated, but its values are never read anywhere in the code, it can be considered unused code. This can be due to some refactoring, copy-pasting, or typing errors.

Unused collections can waste memory usage and slow down the application’s performance. Additionally, they can make the code harder to read and understand, especially for other developers working on the same codebase.

```
function getLength(a, b, c) {
  const strings = [];  // Noncompliant: Array is declared and populated but never read
  strings.push(a);
  strings.push(b);
  strings.push(c);

  return a.length + b.length + c.length;
}
```

Remove unused collections so that the application can run faster and more smoothly. The code becomes cleaner and more efficient, making it easier to read, understand, and maintain.

```
function getLength(a, b, c) {
  return a.length + b.length + c.length;
}
```

resources

Documentation

- MDN web docs - Indexed collections

- MDN web docs - Keyed collections

Searching OS commands in PATH is security-sensitive

Clave: javascript:S4036

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When executing an OS command and unless you specify the full path to the executable, then the locations in your application’s `PATH` environment variable will be searched for the executable. That search could leave an opening for an attacker if one of the elements in `PATH` is a directory under his control.

assess_the_problem

Ask Yourself Whether

- The directories in the PATH environment variable may be defined by not trusted entities.

There is a risk if you answered yes to this question.

Sensitive Code Example

```
const cp = require('child_process');
cp.exec('file.exe'); // Sensitive
```

how_to_fix

Recommended Secure Coding Practices

Fully qualified/absolute path should be used to specify the OS command to execute.

Compliant Solution

```
const cp = require('child_process');
cp.exec('/usr/bin/file.exe'); // Compliant
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-426 - Untrusted Search Path
- CWE - CWE-427 - Uncontrolled Search Path Element

default

When executing an OS command and unless you specify the full path to the executable, then the locations in your application’s `PATH` environment variable will be searched for the executable. That search could leave an opening for an attacker if one of the elements in `PATH` is a directory under his control.

Ask Yourself Whether

- The directories in the PATH environment variable may be defined by not trusted entities.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Fully qualified/absolute path should be used to specify the OS command to execute.

Sensitive Code Example

```
const cp = require('child_process');
cp.exec('file.exe'); // Sensitive
```

Compliant Solution

```
const cp = require('child_process');
cp.exec('/usr/bin/file.exe'); // Compliant
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-426 - Untrusted Search Path
- CWE - CWE-427 - Uncontrolled Search Path Element

Array-mutating methods should not be used misleadingly

Clave: javascript:S4043

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Array copying methods and mutating methods
- MDN web docs - Array.prototype.reverse()
- MDN web docs - Array.prototype.sort()
- MDN web docs - Array.prototype.toReversed()
- MDN web docs - Array.prototype.toSorted()
- MDN web docs - Spread syntax (...)
- MDN web docs - Array.prototype.slice()

root_cause

In JavaScript, some `Array` methods do not mutate the existing array that the method was called on, but instead return a new array. Other methods mutate the array, and their return value differs depending on the method.

`reverse` and `sort` are mutating methods and, in addition, return the altered version. This rule raises an issue when the return values of these methods are assigned, which could lead maintainers to overlook the fact that the original array has been modified.

```
const reversed = a.reverse(); // Noncompliant: mutating method, no need to assign return value
const sorted = b.sort(); // Noncompliant: mutating method, no need to assign return value
```

Remove the assignment, so that the intent of mutating the original array is clear.

```
a.reverse();
b.sort();
```

Or use non-mutating alternatives `toReversed` and `toSorted`.

```
const reversed = a.toReversed();
const sorted = b.toSorted();
```

Alternatively, change a mutating method into a non-mutating alternative using the spread syntax (…).

```
const reversed = [...a].reverse();
const sorted = [...b].sort();
```

Or `slice()` to create a copy first.

```
const reversed = a.slice().reverse();
const sorted = b.slice().sort();
```

Sparse arrays should not be created with extra commas

Clave: javascript:S4140

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A sparse array is an array in which the elements do not occupy a contiguous range of indices. In other words, there are gaps (or "holes") between the elements in the array, where some indices have no corresponding value assigned to them.

Including an extra comma in an array literal signifies an empty slot in the array, thus creating a sparse array. An empty slot is not the same as a slot filled with the `undefined` value. In some operations, empty slots behave as if they are filled with `undefined` but are skipped in others.

While this is a well-defined behavior, it can be misleading and raises suspicions about the original intent: an extra comma was intentionally inserted, or perhaps the developer meant to insert the missing value but forgot.

```
let a = [1, , 3, 6, 9]; // Noncompliant: Extra comma maybe denoting an oversight
```

You should either remove the extra comma if this was a mistake or add the value you meant to insert initially.

```
let a = [1, 3, 6, 9];
```

Alternatively, if the additional comma was intended, this should be made explicit with an `undefined` element.

```
let a = [1, undefined, 3, 6, 9];
```

resources

Documentation

- MDN web docs - Sparse arrays

Collection elements should not be replaced unconditionally

Clave: javascript:S4143

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Storing a value inside a collection at a given key or index and then unconditionally overwriting it without reading the initial value is a case of a "dead store".

This rule detects repeatedly adding an element at the same index or key in a collection or adding identical elements to a set.

```
fruits[1] = "banana";
fruits[1] = "apple";  // Noncompliant

myMap.set("key", 1);
myMap.set("key", 2); // Noncompliant

mySet.add(1);
mySet.add(1); // Noncompliant
```

This practice is redundant and will cause confusion for the reader. More importantly, it is often an error and not what the developer intended to do.

Functions should not have identical implementations

Clave: javascript:S4144

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Two functions having the same implementation are suspicious. It might be that something else was intended. Or the duplication is intentional, which becomes a maintenance burden.

```
function calculateCode() {
  doTheThing();
  doOtherThing();
  return code;
}

function getName() {  // Noncompliant: duplicates calculateCode
  doTheThing();
  doOtherThing();
  return code;
}
```

If the identical logic is intentional, the code should be refactored to avoid duplication. For example, by having both functions call the same function or by having one implementation invoke the other.

```
function calculateCode() {
  doTheThing();
  doOtherThing();
  return code;
}

function getName() { // Intent is clear
  return calculateCode();
}
```

Exceptions

- Functions with fewer than 3 lines are ignored.
- This rule does not apply to function expressions and arrow functions because they don't have explicit names and are often used in a way where refactoring is not applicable.

```
list.map((item) => ({
  name: item.name,
  address: item.address,
  country: item.country
}));
```

Empty collections should not be accessed or iterated

Clave: javascript:S4158

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

When a collection is empty with absolute certainty, it makes no sense to access or iterate it. Doing so can lead to unexpected behavior or errors in the code. The most common cause is that population was accidentally omitted or removed.

```
const strings = [];

if (strings.includes("foo")) {}  // Noncompliant: strings is always empty

for (const str of strings) {}  // Noncompliant

strings.forEach(str => doSomething(str)); // Noncompliant
```

Make sure your code provides some way to populate the collection if their elements are to be accessed.

```
const strings = [];

strings.push("foo");

if (strings.includes("foo")) {}

for (const str of strings) {}

strings.forEach(str => doSomething(str));
```

resources

Documentation

- MDN web docs - Array
- MDN web docs - Iterable protocol

Assignments should not be redundant

Clave: javascript:S4165

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The transitive property says that if `a == b` and `b == c`, then `a == c`. In such cases, there’s no point in assigning `a` to `c` or vice versa because they’re already equivalent.

This rule raises an issue when an assignment is useless because the assigned-to variable already holds the value on all execution paths.

Noncompliant code example

```
a = b;
c = a;
b = c; // Noncompliant: c and b are already the same
```

Compliant solution

```
a = b;
c = a;
```

Getters and setters should access the expected fields

Clave: javascript:S4275

Severidad: CRITICAL

Estado: ☑ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - get
- MDN web docs - set
- MDN web docs - Private class features

root_cause

Getters and setters provide a way to enforce encapsulation by providing methods that give controlled access to class fields. However, in classes with multiple fields, it is not unusual that copy and paste is used to quickly create the needed getters and setters, which can result in the wrong field being accessed by a getter or setter.

This rule raises an issue in the following cases:

- A setter does not update the field with the corresponding name (if it exists).
- A getter:
  - does not return any value
  - does not access the field with the corresponding name (if it exists).

Underscore prefixes for fields are supported, so `setX()` can assign a value to `_x`.

The following type of getters and setters are supported:

- `getX()` and `setX()`

```
class A {
  #y: number = 0;
  setY(val: number) { // Noncompliant: field '#y' is not updated
  }
}
```

```
class A {
  #y: number = 0;
  setY(val: number) {
    this.#y = val;
  }
}
```

- `get x()` and `set x()`

```
class A {
  _x: number = 0;
  #y: number = 0;

  get x() { // Noncompliant: field '_x' is not used in the return value
    return this.#y;
  }

  get y() { // Noncompliant: method may not return any value
    if (condition) {
      return #y;
    }
  }
}
```

```
class A {
  _x: number = 0;
  #y: number = 0;

  get x() {
    return this._x;
  }
  get y() {
    if (condition) {
      return #y;
    }
    return 1;
  }
}
```

- getters and setters defined with `Object.defineProperty()`

```
let x = 0;
let y = 0;
Object.defineProperty(o, 'x', {
  get() { // Noncompliant: variable 'x' is not used in the return value
    return y;
  }
});
```

```
let x = 0;
let y = 0;
Object.defineProperty(o, 'x', {
  get() {
    return x;
  }
});
```

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive

Clave: javascript:S5122

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

default

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-0269
- CVE-2017-14460

Same origin policy in browsers prevents, by default and for security-reasons, a javascript frontend to perform a cross-origin HTTP request to a resource that has a different origin (domain, protocol, or port) from its own. The requested target can append additional HTTP headers in response, called CORS, that act like directives for the browser and change the access control policy / relax the same origin policy.

Ask Yourself Whether

- You don’t trust the origin specified, example: `Access-Control-Allow-Origin: untrustedwebsite.com`.
- Access control policy is entirely disabled: `Access-Control-Allow-Origin: *`
- Your access control policy is dynamically defined by a user-controlled input like `origin` header.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- The `Access-Control-Allow-Origin` header should be set only for a trusted origin and for specific resources.
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the `Origin` header content without any checks).

Sensitive Code Example

nodejs http built-in module:

```
const http = require('http');
const srv = http.createServer((req, res) => {
  res.writeHead(200, { 'Access-Control-Allow-Origin': '*' }); // Sensitive
  res.end('ok');
});
srv.listen(3000);
```

Express.js framework with cors middleware:

```
const cors = require('cors');

let app1 = express();
app1.use(cors()); // Sensitive: by default origin is set to *

let corsOptions = {
  origin: '*' // Sensitive
};

let app2 = express();
app2.use(cors(corsOptions));
```

User-controlled origin:

```
function (req, res) {
  const origin = req.headers.origin;
  res.setHeader('Access-Control-Allow-Origin', origin); // Sensitive
};
```

Compliant Solution

nodejs http built-in module:

```
const http = require('http');
const srv = http.createServer((req, res) => {
  res.writeHead(200, { 'Access-Control-Allow-Origin': 'trustedwebsite.com' }); // Compliant
  res.end('ok');
});
srv.listen(3000);
```

Express.js framework with cors middleware:

```
const cors = require('cors');

let corsOptions = {
  origin: 'trustedwebsite.com' // Compliant
};

let app = express();
app.use(cors(corsOptions));
```

User-controlled origin validated with an allow-list:

```
function (req, res) {
  const origin = req.headers.origin;

  if (origin === 'trustedwebsite.com') {
    res.setHeader('Access-Control-Allow-Origin', origin);
  }
};
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.mozilla.org - CORS
- developer.mozilla.org - Same origin policy
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP HTML5 Security Cheat Sheet - Cross Origin Resource Sharing
- CWE - CWE-346 - Origin Validation Error
- CWE - CWE-942 - Overly Permissive Cross-domain Whitelist

assess_the_problem

Ask Yourself Whether

- You don't trust the origin specified, example: `Access-Control-Allow-Origin: untrustedwebsite.com`.
- Access control policy is entirely disabled: `Access-Control-Allow-Origin: *`
- Your access control policy is dynamically defined by a user-controlled input like `origin` header.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

nodejs http built-in module:

```
const http = require('http');
const srv = http.createServer((req, res) => {
  res.writeHead(200, { 'Access-Control-Allow-Origin': '*' }); // Sensitive
  res.end('ok');
});
srv.listen(3000);
```

Express.js framework with cors middleware:

```
const cors = require('cors');

let app1 = express();
app1.use(cors()); // Sensitive: by default origin is set to *

let corsOptions = {
  origin: '*' // Sensitive
};

let app2 = express();
app2.use(cors(corsOptions));
```

User-controlled origin:

```
function (req, res) {
  const origin = req.headers.origin;
  res.setHeader('Access-Control-Allow-Origin', origin); // Sensitive
};
```

how_to_fix

Recommended Secure Coding Practices

- The `Access-Control-Allow-Origin` header should be set only for a trusted origin and for specific resources.
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the `Origin` header content without any checks).

Compliant Solution

nodejs http built-in module:

```
const http = require('http');
const srv = http.createServer((req, res) => {
  res.writeHead(200, { 'Access-Control-Allow-Origin': 'trustedwebsite.com' }); // Compliant
  res.end('ok');
});
srv.listen(3000);
```

Express.js framework with cors middleware:

```
const cors = require('cors');

let corsOptions = {
  origin: 'trustedwebsite.com' // Compliant
};

let app = express();
app.use(cors(corsOptions));
```

User-controlled origin validated with an allow-list:

```
function (req, res) {
  const origin = req.headers.origin;

  if (origin === 'trustedwebsite.com') {
    res.setHeader('Access-Control-Allow-Origin', origin);
  }
};
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.mozilla.org - CORS
- developer.mozilla.org - Same origin policy
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP HTML5 Security Cheat Sheet - Cross Origin Resource Sharing
- CWE - CWE-346 - Origin Validation Error
- CWE - CWE-942 - Overly Permissive Cross-domain Whitelist

root_cause

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-0269
- CVE-2017-14460

Same origin policy in browsers prevents, by default and for security-reasons, a javascript frontend to perform a cross-origin HTTP request to a resource that has a different origin (domain, protocol, or port) from its own. The requested target can append additional HTTP headers in response, called CORS, that act like directives for the browser and change the access control policy / relax the same origin policy.

Authorizing an opened window to access back to the originating window is security-sensitive

Clave: javascript:S5148

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

Use `noopener` to prevent untrusted pages from abusing `window.opener`.

Compliant Solution

```
window.open("https://example.com/dangerous", "WindowName", "noopener");
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Reverse Tabnabbing
- CWE - CWE-1022 - Use of Web Link to Untrusted Target with window.opener Access
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- https://mathiasbynens.github.io/rel-noopener/

assess_the_problem

Ask Yourself Whether

- The application opens untrusted external URL.

There is a risk if you answered yes to this question.

Sensitive Code Example

```
window.open("https://example.com/dangerous"); // Sensitive
```

default

A newly opened window having access back to the originating window could allow basic phishing attacks (the `window.opener` object is not `null` and thus `window.opener.location` can be set to a malicious website by the opened page).

For instance, an attacker can put a link (say: "http://example.com/mylink") on a popular website that changes, when opened, the original page to "http://example.com/fake_login". On "http://example.com/fake_login" there is a fake login page which could trick real users to enter their credentials.

Ask Yourself Whether

- The application opens untrusted external URL.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Use `noopener` to prevent untrusted pages from abusing `window.opener`.

Sensitive Code Example

```
window.open("https://example.com/dangerous"); // Sensitive
```

Compliant Solution

```
window.open("https://example.com/dangerous", "WindowName", "noopener");
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Reverse Tabnabbing
- CWE - CWE-1022 - Use of Web Link to Untrusted Target with window.opener Access
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- https://mathiasbynens.github.io/rel-noopener/

root_cause

A newly opened window having access back to the originating window could allow basic phishing attacks (the `window.opener` object is not `null` and thus `window.opener.location` can be set to a malicious website by the opened page).

For instance, an attacker can put a link (say: "http://example.com/mylink") on a popular website that changes, when opened, the original page to "http://example.com/fake_login". On "http://example.com/fake_login" there is a fake login page which could trick real users to enter their credentials.

Disabling auto-escaping in template engines is security-sensitive

Clave: javascript:S5247

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

Enable auto-escaping by default and continue to review the use of inputs in order to be sure that the chosen auto-escaping strategy is the right one.

Compliant Solution

mustache.js template engine:

```
let Mustache = require("mustache");

let rendered = Mustache.render(template, { name: inputName }); // Compliant autoescaping is on by default
```

handlebars.js template engine:

```
const Handlebars = require('handlebars');

let source = "<p>attack {{name}}</p>";
let data = { "name": "<b>Alan</b>" };

let template = Handlebars.compile(source); // Compliant by default noEscape is set to false
```

markdown-it markup language parser:

```
let md = require('markdown-it')(); // Compliant by default html is set to false

let result = md.render('# <b>attack</b>');
```

marked markup language parser:

```
const marked = require('marked');

marked.setOptions({
  renderer: new marked.Renderer()
}); // Compliant by default sanitize is set to true

console.log(marked("# test <b>attack/b>"));
```

kramed markup language parser:

```
let kramed = require('kramed');

let options = {
  renderer: new kramed.Renderer({
    sanitize: true // Compliant
  })
};

console.log(kramed('Attack [xss?](javascript:alert("xss")).', options));
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP Cheat Sheet - XSS Prevention Cheat Sheet
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

root_cause

To reduce the risk of cross-site scripting attacks, templating systems, such as `Twig`, `Django`, `Smarty`, `Groovy's template engine`, allow configuration of automatic variable escaping before rendering templates. When escape occurs, characters that make sense to the browser (eg: <a>) will be transformed/replaced with escaped/sanitized values (eg: & lt;a& gt;).

Auto-escaping is not a magic feature to annihilate all cross-site scripting attacks, it depends on the strategy applied and the context, for example a "html auto-escaping" strategy (which only transforms html characters into html entities) will not be relevant when variables are used in a html attribute because '`:`' character is not escaped and thus an attack as below is possible:

```
<a href="{{ myLink }}">link</a> // myLink = javascript:alert(document.cookie)
<a href="javascript:alert(document.cookie)">link</a> // JS injection (XSS attack)
```

assess_the_problem

Ask Yourself Whether

- Templates are used to render web content and
  - dynamic variables in templates come from untrusted locations or are user-controlled inputs
  - there is no local mechanism in place to sanitize or validate the inputs.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

mustache.js template engine:

```
let Mustache = require("mustache");

Mustache.escape = function(text) {return text;}; // Sensitive

let rendered = Mustache.render(template, { name: inputName });
```

handlebars.js template engine:

```
const Handlebars = require('handlebars');

let source = "<p>attack {{name}}</p>";

let template = Handlebars.compile(source, { noEscape: true }); // Sensitive
```

markdown-it markup language parser:

```
const markdownIt = require('markdown-it');
let md = markdownIt({
  html: true // Sensitive
});

let result = md.render('# <b>attack</b>');
```

marked markup language parser:

```
const marked = require('marked');

marked.setOptions({
  renderer: new marked.Renderer(),
  sanitize: false // Sensitive
});

console.log(marked("# test <b>attack/b>"));
```

kramed markup language parser:

```
let kramed = require('kramed');

var options = {
  renderer: new kramed.Renderer({
    sanitize: false // Sensitive
  })
};
```

default

To reduce the risk of cross-site scripting attacks, templating systems, such as `Twig`, `Django`, `Smarty`, `Groovy's template engine`, allow configuration of automatic variable escaping before rendering templates. When escape occurs, characters that make sense to the browser (eg: <a>) will be transformed/replaced with escaped/sanitized values (eg: & lt;a& gt;).

Auto-escaping is not a magic feature to annihilate all cross-site scripting attacks, it depends on the strategy applied and the context, for example a "html auto-escaping" strategy (which only transforms html characters into html entities) will not be relevant when variables are used in a html attribute because '`:`' character is not escaped and thus an attack as below is possible:

```
<a href="{{ myLink }}">link</a> // myLink = javascript:alert(document.cookie)
<a href="javascript:alert(document.cookie)">link</a> // JS injection (XSS attack)
```

Ask Yourself Whether

- Templates are used to render web content and
  - dynamic variables in templates come from untrusted locations or are user-controlled inputs
  - there is no local mechanism in place to sanitize or validate the inputs.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Enable auto-escaping by default and continue to review the use of inputs in order to be sure that the chosen auto-escaping strategy is the right one.

Sensitive Code Example

mustache.js template engine:

```
let Mustache = require("mustache");

Mustache.escape = function(text) {return text;}; // Sensitive

let rendered = Mustache.render(template, { name: inputName });
```

handlebars.js template engine:

```
const Handlebars = require('handlebars');

let source = "<p>attack {{name}}</p>";

let template = Handlebars.compile(source, { noEscape: true }); // Sensitive
```

markdown-it markup language parser:

```
const markdownIt = require('markdown-it');
let md = markdownIt({
  html: true // Sensitive
});

let result = md.render('# <b>attack</b>');
```

marked markup language parser:

```
const marked = require('marked');

marked.setOptions({
  renderer: new marked.Renderer(),
  sanitize: false // Sensitive
});

console.log(marked("# test <b>attack/b>"));
```

kramed markup language parser:

```
let kramed = require('kramed');

var options = {
  renderer: new kramed.Renderer({
    sanitize: false // Sensitive
  })
};
```

Compliant Solution

mustache.js template engine:

```
let Mustache = require("mustache");

let rendered = Mustache.render(template, { name: inputName }); // Compliant autoescaping is on by default
```

handlebars.js template engine:

```
const Handlebars = require('handlebars');

let source = "<p>attack {{name}}</p>";
let data = { "name": "<b>Alan</b>" };

let template = Handlebars.compile(source); // Compliant by default noEscape is set to false
```

markdown-it markup language parser:

```
let md = require('markdown-it')(); // Compliant by default html is set to false

let result = md.render('# <b>attack</b>');
```

marked markup language parser:

```
const marked = require('marked');

marked.setOptions({
  renderer: new marked.Renderer()
}); // Compliant by default sanitize is set to true

console.log(marked("# test <b>attack/b>"));
```

kramed markup language parser:

```
let kramed = require('kramed');

let options = {
  renderer: new kramed.Renderer({
    sanitize: true // Compliant
  })
};

console.log(kramed('Attack [xss?](javascript:alert("xss")).', options));
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP Cheat Sheet - XSS Prevention Cheat Sheet
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

HTML elements should have a valid language attribute

Clave: javascript:S5254

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

introduction

HTML documents should have a valid IETF’s BCP 47 `lang` attribute.

how_to_fix

Add a `lang` attribute with a valid IETF BCP 47 value.

Noncompliant code example

```
<html></html>
```

Compliant solution

```
<html lang="en"></html>
```

resources

Documentation

- W3C, Language tags in HTML and XML

root_cause

Screen readers require a specified language to function correctly. Without it, they default to the user’s set language, causing issues for multilingual users. Specifying a valid language ensures correct pronunciation and a less confusing experience.

Tables should have headers

Clave: javascript:S5256

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- WCAG2, 1.3.1 - Info and Relationships
- WCAG2, H51 - Using table markup to present tabular information

how_to_fix

The first `<tr>` of the table should contain `<th>` elements, with the appropriate description of what the data in those columns represents.

Going the extra mile

Headers should be properly associated with the corresponding `<td>` cells by using either a `scope` attribute or `headers` and `id` attributes. See W3C WAI Web Accessibility Tutorials for more information.

Noncompliant code example

```
<table> <!-- Noncompliant -->
  <tr>
    <td>Name</td>
    <td>Age</td>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>24</td>
  </tr>
  <tr>
    <td>Alice Doe</td>
    <td>54</td>
  </tr>
</table>
```

Compliant solution

```
<table>
  <tr>
    <th scope="col">Name</th>
    <th scope="col">Age</th>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>24</td>
  </tr>
  <tr>
    <td>Alice Doe</td>
    <td>54</td>
  </tr>
</table>
```

root_cause

Table headers are essential to enhance the accessibility of a table’s data, particularly for assistive technologies like screen readers. These headers provide the necessary context to transform data into information. Without headers, users get rapidly lost in the flow of data.

This rule raises an issue whenever it detects a table that does not have a full row or column with `<th>` elements.

Exceptions

No issue will be raised on `<table>` used for layout purpose, i.e. when it contains a `role` attribute set to `"presentation"` or `"none"`.

```
<table role="presentation">
  <tr>
    <td>Name</td>
    <td>Age</td>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>42</td>
  </tr>
</table>
```

Note that using <table> for layout purpose is a bad practice.

No issue will be raised on `<table>` containing an `aria-hidden` attribute set to `"true"`.

```
<table aria-hidden="true">
  <tr>
    <td>Name</td>
    <td>Age</td>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>42</td>
  </tr>
</table>
```

HTML "" should not be used for layout purposes

Clave: javascript:S5257

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

- **Document and website structure**

root_cause

HTML <table> elements should not be used for layout purposes as it can confuse screen readers. It is recommended to use CSS instead.

This rule raises an issue on every `<table>` element containing a `role` attribute set to "`presentation`" or "`none`", which is how W3C recommends marks layout tables.

Noncompliant code example

```
<table role="presentation"><!-- Noncompliant -->
<!-- ... -->
</table>
```

Table cells should reference their headers

Clave: javascript:S5260

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

resources

- **WCAG2, 1.3.1 - Info and Relationships**
- **WCAG2, H43 - Using id and headers attributes to associate data cells with header cells in data tables**

root_cause

If a `<td>` cell has a `headers` attribute, it should reference only IDs of headers in the same column and row.

Note that it is usually better to use `scope` attributes of `<th>` headers instead of `headers` attribute. `headers` attribute requires you to list every corresponding `<th>` header's `id`, which is error-prone and makes the code less maintainable. See W3C WAI Web Accessibility Tutorials for more information.

If your table is too complex, it might be better to split it into multiple small tables as it improves both readability and maintainability.

This rule raises an issue when the `headers` attribute of a `<td>` cell contains IDs which don't belong to a header in the same row or column.

Noncompliant code example

```
<table border="1">
<caption>
    Rental price
</caption>
<thead>
    <tr>
        <td></td>
        <th id="small" scope="col">
            Small car
        </th>
        <th id="big" scope="col">
            Big Car
        </th>
    </tr>
</thead>
```

```
<tbody>
    <tr>
        <th id="paris" class="span" colspan="3" scope="colgroup">
            Paris
        </th>
        </tr>
        <tr>
        <th headers="paris" id="day1">
            1 day
        </th>
        <td headers="paris day1 big"> <!-- Noncompliant, referencing the column "big" instead of "small" -->
            11 euros
        </td>
        <td headers="berlin day1 big"> <!-- Noncompliant, there is no header with id "berlin" -->
            50 euros
        </td>
        </tr>
    </tr>
</tbody>
</table>
```

"" tags should provide an alternative content

Clave: javascript:S5264

Severidad: MINOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

The `<object>` HTML element represents an external resource, which can be treated as an image, a nested browsing context, or a resource to be handled by a plugin. The element’s children are the fallback content. This allows multiple object elements to be nested inside each other, targeting multiple user agents with different capabilities, with the user agent picking the first one it supports.

Assistive technologies, such as screen readers, will not be able to render `<object>` elements, in such cases it is the content of the `<object>` which is provided to the user. This alternative content needs to be accessible or the screen readers won’t be able to use it. For example, if an `<img>` is used it must contain an `alt` attribute.

This rule raises an issue when an `<object>` tag does not have any alternative content.

```
<object></object> <!-- Noncompliant -->

<object>
  <object></object> <!-- Noncompliant -->
</object>
```

Provide alternative content to `<object>` elements.

```
<object>This application shows the simulation of two particles colliding</object>

<object>
  <img src="flower.png" alt="Flower growing in a pot" />
</object>

<object>
  <object>
    This application shows the simulation of two particles colliding
  </object>
</object>
```

resources

- **WCAG2, H53** - Using the body of the object element
- **WCAG2, 1.1.1** - Non-text Content
- **WCAG2, 1.2.3** - Audio Description or Media Alternative (Prerecorded)
- **WCAG2, 1.2.8** - Media Alternative (Prerecorded)

Related rules

- Web:ImgWithoutAltCheck

Using publicly writable directories is security-sensitive

Clave: javascript:S5443

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- Files are read from or written into a publicly writable folder
- The application creates files with predictable names into a publicly writable folder

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const fs = require('fs');

let tmp_file = "/tmp/temporary_file"; // Sensitive
fs.readFile(tmp_file, 'utf8', function (err, data) {
  // ...
});
```

```
const fs = require('fs');

let tmp_dir = process.env.TMPDIR; // Sensitive
fs.readFile(tmp_dir + "/temporary_file", 'utf8', function (err, data) {
  // ...
});
```

default

Operating systems have global directories where any user has write access. Those folders are mostly used as temporary storage areas like `/tmp` in Linux based systems. An application manipulating files from these folders is exposed to race conditions on filenames: a malicious user can try to create a file with a predictable name before the application does. A successful attack can result in other files being accessed, modified, corrupted or deleted. This risk is even higher if the application runs with elevated permissions.

In the past, it has led to the following vulnerabilities:

- CVE-2012-2451
- CVE-2015-1838

This rule raises an issue whenever it detects a hard-coded path to a publicly writable directory like `/tmp` (see examples bellow). It also detects access to environment variables that point to publicly writable directories, e.g., `TMP` and `TMPDIR`.

- `/tmp`
- `/var/tmp`
- `/usr/tmp`
- `/dev/shm`
- `/dev/mqueue`
- `/run/lock`

- `/var/run/lock`
- `/Library/Caches`
- `/Users/Shared`
- `/private/tmp`
- `/private/var/tmp`
- `\Windows\Temp`
- `\Temp`
- `\TMP`

Ask Yourself Whether

- Files are read from or written into a publicly writable folder
- The application creates files with predictable names into a publicly writable folder

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use a dedicated sub-folder with tightly controlled permissions
- Use secure-by-design APIs to create temporary files. Such API will make sure:
  - The generated filename is unpredictable
  - The file is readable and writable only by the creating user ID
  - The file descriptor is not inherited by child processes
  - The file will be destroyed as soon as it is closed

Sensitive Code Example

```
const fs = require('fs');

let tmp_file = "/tmp/temporary_file"; // Sensitive
fs.readFile(tmp_file, 'utf8', function (err, data) {
  // ...
});
```

```
const fs = require('fs');

let tmp_dir = process.env.TMPDIR; // Sensitive
fs.readFile(tmp_dir + "/temporary_file", 'utf8', function (err, data) {
  // ...
});
```

Compliant Solution

```
const tmp = require('tmp');

const tmpobj = tmp.fileSync(); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-377 - Insecure Temporary File
- CWE - CWE-379 - Creation of Temporary File in Directory with Incorrect Permissions
- OWASP, Insecure Temporary File
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

how_to_fix

Recommended Secure Coding Practices

- Use a dedicated sub-folder with tightly controlled permissions
- Use secure-by-design APIs to create temporary files. Such API will make sure:
  - The generated filename is unpredictable
  - The file is readable and writable only by the creating user ID
  - The file descriptor is not inherited by child processes
  - The file will be destroyed as soon as it is closed

Compliant Solution

```
const tmp = require('tmp');

const tmpobj = tmp.fileSync(); // Compliant
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-377 - Insecure Temporary File
- CWE - CWE-379 - Creation of Temporary File in Directory with Incorrect Permissions
- OWASP, Insecure Temporary File
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

root_cause

Operating systems have global directories where any user has write access. Those folders are mostly used as temporary storage areas like `/tmp` in Linux based systems. An application manipulating files from these folders is exposed to race conditions on filenames: a malicious user can try to create a file with a predictable name before the application does. A successful attack can result in other files being accessed, modified, corrupted or deleted. This risk is even higher if the application runs with elevated permissions.

In the past, it has led to the following vulnerabilities:

- CVE-2012-2451
- CVE-2015-1838

This rule raises an issue whenever it detects a hard-coded path to a publicly writable directory like `/tmp` (see examples bellow). It also detects access to environment variables that point to publicly writable directories, e.g., `TMP` and `TMPDIR`.

- `/tmp`
- `/var/tmp`
- `/usr/tmp`
- `/dev/shm`
- `/dev/mqueue`
- `/run/lock`
- `/var/run/lock`
- `/Library/Caches`
- `/Users/Shared`
- `/private/tmp`
- `/private/var/tmp`
- `\Windows\Temp`
- `\Temp`

- `\TMP`

Disclosing fingerprints from web application technologies is security-sensitive

Clave: javascript:S5689

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- **Version information is accessible to end users.**
- **Internal systems do not benefit from timely patch management workflows.**

There is a risk if you answered yes to any of these questions.

Sensitive Code Example

In Express.js, version information is disclosed by default in the `x-powered-by` HTTP header:

```
let express = require('express');

let example = express(); // Sensitive

example.get('/', function (req, res) {
  res.send('example')
});
```

root_cause

Disclosure of version information, usually overlooked by developers but disclosed by default by the systems and frameworks in use, can pose a significant security risk depending on the production environement.

Once this information is public, attackers can use it to identify potential security holes or vulnerabilities specific to that version.

Furthermore, if the published version information indicates the use of outdated or unsupported software, it becomes easier for attackers to exploit known vulnerabilities. They can search for published vulnerabilities related to that version and launch attacks that specifically target those vulnerabilities.

how_to_fix

Recommended Secure Coding Practices

In general, it is recommended to keep internal technical information within internal systems to control what attackers know about the underlying architectures. This is known as the "need to know" principle.

The most effective solution is to remove version information disclosure from what end users can see, such as the "x-powered-by" header.
This can be achieved directly through the web application code, server (nginx, apache) or firewalls.

Disabling the server signature provides additional protection by reducing the amount of information available to attackers. Note, however, that this does not provide as much protection as regular updates and patches.
Security by obscurity is the least foolproof solution of all. It should never be the only defense mechanism and should always be combined with other security measures.

Compliant Solution

`x-powered-by` HTTP header should be disabled in **Express.js** with `app.disable`:

```
let express = require('express');

let example = express();
example.disable("x-powered-by");
```

Or with helmet's **hidePoweredBy** middleware:

```
let helmet = require("helmet");

let example = express();
example.use(helmet.hidePoweredBy());
```

See

- OWASP - **Top 10 2021 Category A5 - Security Misconfiguration**
- **OWASP Testing Guide - OTG-INFO-008** - Fingerprint Web Application Framework
- OWASP - **Top 10 2017 Category A6 - Security Misconfiguration**
- CWE - **CWE-200 - Information Exposure**

default

Disclosure of version information, usually overlooked by developers but disclosed by default by the systems and frameworks in use, can pose a significant security risk depending on the production environement.

Once this information is public, attackers can use it to identify potential security holes or vulnerabilities specific to that version.

Furthermore, if the published version information indicates the use of outdated or unsupported software, it becomes easier for attackers to exploit known vulnerabilities. They can search for published vulnerabilities related to that version and launch attacks that specifically target those vulnerabilities.

Ask Yourself Whether

- Version information is accessible to end users.
- Internal systems do not benefit from timely patch management workflows.

There is a risk if you answered yes to any of these questions.

Recommended Secure Coding Practices

In general, it is recommended to keep internal technical information within internal systems to control what attackers know about the underlying architectures. This is known as the "need to know" principle.

The most effective solution is to remove version information disclosure from what end users can see, such as the "x-powered-by" header.
This can be achieved directly through the web application code, server (nginx, apache) or firewalls.

Disabling the server signature provides additional protection by reducing the amount of information available to attackers. Note, however, that this does not provide as much protection as regular updates and patches.
Security by obscurity is the least foolproof solution of all. It should never be the only defense mechanism and should always be combined with other security measures.

Sensitive Code Example

In **Express.js**, version information is disclosed by default in the `x-powered-by` HTTP header:

```
let express = require('express');

let example = express(); // Sensitive

example.get('/', function (req, res) {
  res.send('example')
});
```

Compliant Solution

`x-powered-by` HTTP header should be disabled in **Express.js** with `app.disable`:

```
let express = require('express');

let example = express();
example.disable("x-powered-by");
```

Or with helmet’s **hidePoweredBy** middleware:

```
let helmet = require("helmet");

let example = express();
example.use(helmet.hidePoweredBy());
```

See

- OWASP - **Top 10 2021 Category A5 - Security Misconfiguration**
- **OWASP Testing Guide - OTG-INFO-008 -** Fingerprint Web Application Framework
- OWASP - **Top 10 2017 Category A6 - Security Misconfiguration**
- CWE - **CWE-200 - Information Exposure**

Administration services access should be restricted to specific IP addresses

Clave: javascript:S6321

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- **AWS Documentation -** Security groups for your VPC
- **Azure Documentation -** Network security groups
- **GCP Documentation -** Firewalls

Standards

- CWE - **CWE-284 - Improper Access Control**
- OWASP - **Top 10 2021 Category A1 - Broken Access Control**
- OWASP - **Top 10 2017 Category A3 - Sensitive Data Exposure**

root_cause

Cloud platforms such as AWS, Azure, or GCP support virtual firewalls that can be used to restrict access to services by controlling inbound and outbound traffic.
Any firewall rule allowing traffic from all IP addresses to standard network ports on which administration services traditionally listen, such as 22 for SSH, can expose these services to exploits and unauthorized access.

What is the potential impact?

Like any other service, administration services can contain vulnerabilities. Administration services run with elevated privileges and thus a vulnerability could have a high impact on the system.

Additionally, credentials might be leaked through phishing or similar techniques. Attackers who are able to reach the services could use the credentials to log in to the system.

how_to_fix

It is recommended to restrict access to remote administration services to only trusted IP addresses. In practice, trusted IP addresses are those held by system administrators or those of bastion-like servers.

Noncompliant code example

For aws-cdk-lib.aws_ec2.Instance and other constructs that support a `connections` attribute:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const instance = new ec2.Instance(this, "default-own-security-group",{
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    instanceName: "test-instance"
})

instance.connections.allowFrom(
    ec2.Peer.anyIpv4(), // Noncompliant
    ec2.Port.tcp(22),
    /*description*/ "Allows SSH from all IPv4"
)
```

For aws-cdk-lib.aws_ec2.SecurityGroup

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const securityGroup = new ec2.SecurityGroup(this, "custom-security-group", {
    vpc: vpc
})

securityGroup.addIngressRule(
    ec2.Peer.anyIpv4(), // Noncompliant
    ec2.Port.tcpRange(1, 1024)
)
```

For aws-cdk-lib.aws_ec2.CfnSecurityGroup

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnSecurityGroup(
    this,
    "cfn-based-security-group", {
        groupDescription: "cfn based security group",
        groupName: "cfn-based-security-group",
        vpcId: vpc.vpcId,
        securityGroupIngress: [
            {
                ipProtocol: "6",
                cidrIp: "0.0.0.0/0", // Noncompliant
                fromPort: 22,
                toPort: 22
            }
        ]
    }
)
```

For aws-cdk-lib.aws_ec2.CfnSecurityGroupIngress

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnSecurityGroupIngress( // Noncompliant
    this,
    "ingress-all-ip-tcp-ssh", {
        ipProtocol: "tcp",
        cidrIp: "0.0.0.0/0",
        fromPort: 22,
        toPort: 22,
        groupId: securityGroup.attrGroupId
})
```

Compliant solution

For aws-cdk-lib.aws_ec2.Instance and other constructs that support a `connections` attribute:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const instance = new ec2.Instance(this, "default-own-security-group",{
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    instanceName: "test-instance"
})

instance.connections.allowFrom(
    ec2.Peer.ipv4("192.0.2.0/24"),
    ec2.Port.tcp(22),
    /*description*/ "Allows SSH from a trusted range"
)
```

For aws-cdk-lib.aws_ec2.SecurityGroup

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const securityGroup3 = new ec2.SecurityGroup(this, "custom-security-group", {
    vpc: vpc
})

securityGroup3.addIngressRule(
    ec2.Peer.anyIpv4(),
    ec2.Port.tcpRange(1024, 1048)
)
```

For aws-cdk-lib.aws_ec2.CfnSecurityGroup

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnSecurityGroup(
    this,
    "cfn-based-security-group", {
        groupDescription: "cfn based security group",
        groupName: "cfn-based-security-group",
        vpcId: vpc.vpcId,
        securityGroupIngress: [
            {
                ipProtocol: "6",
                cidrIp: "192.0.2.0/24",
                fromPort: 22,
                toPort: 22
            }
        ]
    }
)
```

For aws-cdk-lib.aws_ec2.CfnSecurityGroupIngress

```
new ec2.CfnSecurityGroupIngress(
    this,
    "ingress-all-ipv4-tcp-http", {
        ipProtocol: "6",
        cidrIp: "0.0.0.0/0",
        fromPort: 80,
        toPort: 80,
        groupId: securityGroup.attrGroupId
    }
)
```

Alternation in regular expressions should not contain empty alternatives

Clave: javascript:S6323

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Alternation is used to match a single regular expression out of several possible regular expressions. If one of the alternatives is empty it would match any input, which is most probably a mistake.

Noncompliant code example

```
/Jack|Peter|/.test('John'); // Noncompliant - returns 'true'
/Jack||Peter/.test('John'); // Noncompliant - returns 'true'
```

Compliant solution

```
/Jack|Peter/.test('John'); // returns 'false'
```

Exceptions

One could use an empty alternation to make a regular expression group optional. Rule will not report on such cases.

```
/mandatory(-optional|)/.test('mandatory'); // returns 'true'
/mandatory(-optional|)/.test('mandatory-optional'); // returns 'true'
```

However, if there is a quantifier after the group the issue will be reported as using both `|` and quantifier is redundant.

```
/mandatory(-optional|)?/.test('mandatory'); // Noncompliant - using both `|` inside the group and `?` for the g
```

Regular expressions should not contain control characters

Clave: javascript:S6324

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- **Wikipedia - ASCII**
- **Wikipedia - C0 and C1 control codes**
- **MDN web docs - Character classes**

root_cause

Entries in the ASCII table below code 32 are known as control characters or non-printing characters. As they are not common in JavaScript strings, using these invisible characters in regular expressions is most likely a mistake.

```
const pattern1 = /\x1a/;             // Noncompliant: 1a (23 base 10) is less than 32
const pattern2 = new RegExp('\x1a'); // Noncompliant: 1a (23 base 10) is less than 32
```

Instead, one should only match printable characters in regular expressions.

```
const pattern1 = /\x20/;
const pattern2 = new RegExp('\x20');
```

Regular expression literals should be used when possible

Clave: javascript:S6325

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - Regular expressions
- MDN web docs - `RegExp`

root_cause

Using regular expression literals is recommended over using the `RegExp` constructor calls if the pattern is a literal. Regular expression literals are shorter, more readable, and do not need to be escaped like string literals. They can also be more performant because regular expression literals are compiled only once when the script is loaded.

```
new RegExp(/foo/);
new RegExp('bar');
new RegExp('baz', 'i');
new RegExp("\\d+");
new RegExp(`qux|quuz`);
```

Using the `RegExp` constructor is suitable when the pattern is computed dynamically, for example, when the user provides it. Otherwise, you should prefer the more concise syntax of regular expression literals.

```
/foo/;
/bar/;
/baz/i;
/\d+/;
/qux|quuz/;
new RegExp(`Dear ${title},`);
```

Regular expressions should not contain multiple spaces

Clave: javascript:S6326

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Multiple spaces in a regular expression can make it hard to tell how many spaces should be matched. It’s more readable to use only one space and then indicate with a quantifier how many spaces are expected.

Noncompliant code example

```
const pattern = /Hello,   world!/;
```

Compliant solution

```
const pattern = /Hello, {3}world!/;
```

Using unencrypted SNS topics is security-sensitive

Clave: javascript:S6327

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

It is recommended to encrypt SNS topics that contain sensitive information.

To do so, create a master key and assign the SNS topic to it. Note that this system does not encrypt the following:

- Topic metadata (topic name and attributes)
- Message metadata (subject, message ID, timestamp, and attributes)
- Data protection policy
- Per-topic metrics

Then, make sure that any publishers have the `kms:GenerateDataKey*` and `kms:Decrypt` permissions for the AWS KMS key.

See AWS SNS Key Management Documentation for more information.

Compliant Solution

For `aws_cdk.aws_sns.Topic`

```
import { Topic } from 'aws-cdk-lib/aws-sns';

const encryptionKey = new Key(this, 'exampleKey', {
    enableKeyRotation: true,
});

new Topic(this, 'exampleTopic', {
    masterKey: encryptionKey
});
```

For `aws_cdk.aws_sns.CfnTopic`

```
import { CfnTopic } from 'aws-cdk-lib/aws-sns';

const encryptionKey = new Key(this, 'exampleKey', {
    enableKeyRotation: true,
});

cfnTopic = new CfnTopic(this, 'exampleCfnTopic', {
    kmsMasterKeyId: encryptionKey.keyId
});
```

See

- AWS Documentation - Encryption at rest
- Encrypting messages published to Amazon SNS with AWS KMS
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of

information at rest.

root_cause

Amazon Simple Notification Service (SNS) is a managed messaging service for application-to-application (A2A) and application-to-person (A2P) communication. SNS topics allows publisher systems to fanout messages to a large number of subscriber systems. Amazon SNS allows to encrypt messages when they are received. In the case that adversaries gain physical access to the storage medium or otherwise leak a message they are not able to access the data.

default

Amazon Simple Notification Service (SNS) is a managed messaging service for application-to-application (A2A) and application-to-person (A2P) communication. SNS topics allows publisher systems to fanout messages to a large number of subscriber systems. Amazon SNS allows to encrypt messages when they are received. In the case that adversaries gain physical access to the storage medium or otherwise leak a message they are not able to access the data.

Ask Yourself Whether

- The topic contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to encrypt SNS topics that contain sensitive information.

To do so, create a master key and assign the SNS topic to it. Note that this system does not encrypt the following:

- Topic metadata (topic name and attributes)
- Message metadata (subject, message ID, timestamp, and attributes)
- Data protection policy
- Per-topic metrics

Then, make sure that any publishers have the `kms:GenerateDataKey*` and `kms:Decrypt` permissions for the AWS KMS key.

See AWS SNS Key Management Documentation for more information.

Sensitive Code Example

For `aws_cdk.aws_sns.Topic`

```
import { Topic } from 'aws-cdk-lib/aws-sns';

new Topic(this, 'exampleTopic'); // Sensitive
```

For `aws_cdk.aws_sns.CfnTopic`

```
import { Topic, CfnTopic } from 'aws-cdk-lib/aws-sns';

new CfnTopic(this, 'exampleCfnTopic'); // Sensitive
```

Compliant Solution

For `aws_cdk.aws_sns.Topic`

```
import { Topic } from 'aws-cdk-lib/aws-sns';

const encryptionKey = new Key(this, 'exampleKey', {
    enableKeyRotation: true,
});

new Topic(this, 'exampleTopic', {
    masterKey: encryptionKey
});
```

For `aws_cdk.aws_sns.CfnTopic`

```
import { CfnTopic } from 'aws-cdk-lib/aws-sns';

const encryptionKey = new Key(this, 'exampleKey', {
    enableKeyRotation: true,
});

cfnTopic = new CfnTopic(this, 'exampleCfnTopic', {
    kmsMasterKeyId: encryptionKey.keyId
});
```

See

- **AWS Documentation - Encryption at rest**
- **Encrypting messages published to Amazon SNS with AWS KMS**
- **CWE - CWE-311 - Missing Encryption of Sensitive Data**
- **STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.**

assess_the_problem

Ask Yourself Whether

- **The topic contains sensitive data that could cause harm when leaked.**
- **There are compliance requirements for the service to store data encrypted.**

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For `aws_cdk.aws_sns.Topic`

```
import { Topic } from 'aws-cdk-lib/aws-sns';

new Topic(this, 'exampleTopic'); // Sensitive
```

For `aws_cdk.aws_sns.CfnTopic`

```
import { Topic, CfnTopic } from 'aws-cdk-lib/aws-sns';

new CfnTopic(this, 'exampleCfnTopic'); // Sensitive
```

Replacement strings should reference existing regular expression groups

Clave: javascript:S6328

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `String.prototype.replace()`
- MDN web docs - `String.prototype.replaceAll()`
- MDN web docs - Regular expressions
- MDN web docs - Capturing group: `(...)`

root_cause

In JavaScript, the `String.prototype.replace()` method is used to replace parts of a string with new substrings. It allows you to perform simple to complex string replacements based on either a static string or a regular expression pattern.

When the first argument is a regular expression, the method will use the regular expression to search for matches within the original string and then replace those matches with the specified replacement. If the second argument is a string, the method will use it as the static replacement for the matched substrings found by the regular expression.

Within the replacement string, the function supports special placeholders to insert the matched values of capturing groups from the regular expression:

- The `$n` syntax allows you to reference capturing groups by their numerical index. The number `n` corresponds to the order in which the capturing group appears in the regular expression, starting from 1 for the first capturing group.
- The `$<Name>` syntax allows you to reference capturing groups by their name. Instead of using numerical indices, you can assign a name to a capturing group using `?<Name>` within the regular expression.

If the second argument of `String.prototype.replace()` references non-existing groups (capturing groups that do not exist in the regular expression), the behavior of the replacement will depend on the specific references made. It won’t cause an error, but the replacement will not be based on any captured values, potentially leading to unexpected results in the replaced string:

- If the replacement string contains references like `$1`, `$2`, etc., to capturing groups that don’t exist in the regular expression, those references will be treated as literals. In other words, the `$n` will be replaced with the literal text `$n` itself.
- If the replacement string contains references like `$<Name>`, they will also be treated as literals, but only if there are no named captures in the regular expression; otherwise, they will be replaced with the empty string.

This rule checks that all referenced groups exist when replacing a pattern with a replacement string using `String.prototype.replace()` or `String.prototype.replaceAll()` methods.

```
const str = 'John Doe';
console.log(str.replace(/(\w+)\s(\w+)/, '$1, $0 $1')); // Noncompliant: index is 1-based, '$0' does not exist,
console.log(str.replace(/(?<firstName>\w+)\s(?<lastName>\w+)/, '$<surname>, $<firstName> $<surname>')); // Nonc
```

Always check your regular expression and replacement string to ensure they properly reference existing capturing groups, most specifically, the latter references capturing groups existing in the former.

```
const str = 'John Doe';
console.log(str.replace(/(\w+)\s(\w+)/, '$2, $1 $2'));
console.log(str.replace(/(?<firstName>\w+)\s(?<lastName>\w+)/, '$<lastName>, $<firstName> $<lastName>'));
```

Allowing public network access to cloud resources is security-sensitive

Clave: javascript:S6329

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

Avoid publishing cloud services on the Internet unless they are intended to be publicly accessible, such as customer portals or e-commerce sites.

Use private networks (and associated private IP addresses) and VPC peering or other secure communication tunnels to communicate with other cloud components.

The goal is to prevent the component from intercepting traffic coming in via the public IP address. If the cloud resource does not support the absence of a public IP address, assign a public IP address to it, but do not create listeners for the public IP address.

Compliant Solution

For **aws-cdk-lib.aws_ec2.Instance** and similar constructs:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.Instance(
    this,
    "example", {
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    vpcSubnets: {subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS}
})
```

For **aws-cdk-lib.aws_ec2.CfnInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnInstance(this, "example", {
    instanceType: "t2.micro",
    imageId: "ami-0ea0f26a6d50850c5",
    networkInterfaces: [
        {
            deviceIndex: "0",
            associatePublicIpAddress: false,
            deleteOnTermination: true,
            subnetId: vpc.selectSubnets({subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS}).subnetIds[0]
        }
    ]
})
```

For **aws-cdk-lib.aws_dms.CfnReplicationInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new dms.CfnReplicationInstance(
    this, "example", {
    replicationInstanceClass: "dms.t2.micro",
    allocatedStorage: 5,
    publiclyAccessible: false,
    replicationSubnetGroupIdentifier: subnetGroup.replicationSubnetGroupIdentifier,
    vpcSecurityGroupIds: [vpc.vpcDefaultSecurityGroup]
})
```

For **aws-cdk-lib.aws_rds.CfnDBInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const rdsSubnetGroupPrivate = new rds.CfnDBSubnetGroup(this, "example",{
    dbSubnetGroupDescription: "Subnets",
    dbSubnetGroupName: "privateSn",
    subnetIds: vpc.selectSubnets({
        subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS
    }).subnetIds
})

new rds.CfnDBInstance(this, "example", {
    engine: "postgres",
    masterUsername: "foobar",
    masterUserPassword: "12345678",
    dbInstanceClass: "db.r5.large",
    allocatedStorage: "200",
    iops: 1000,
    dbSubnetGroupName: rdsSubnetGroupPrivate.ref,
    publiclyAccessible: false,
    vpcSecurityGroups: [sg.securityGroupId]
})
```

See

- AWS Documentation - Amazon EC2 instance IP addressing
- AWS Documentation - Public and private replication instances
- AWS Documentation - VPC Peering
- CWE - CWE-284 - Improper Access Control
- CWE - CWE-668 - Exposure of Resource to Wrong Sphere
- STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.

root_cause

Enabling public network access to cloud resources can affect an organization’s ability to protect its data or internal operations from data theft or disruption.

Depending on the component, inbound access from the Internet can be enabled via:

- a boolean value that explicitly allows access to the public network.
- the assignment of a public IP address.
- database firewall rules that allow public IP ranges.

Deciding to allow public access may happen for various reasons such as for quick maintenance, time saving, or by accident.

This decision increases the likelihood of attacks on the organization, such as:

- data breaches.
- intrusions into the infrastructure to permanently steal from it.
- and various malicious traffic, such as DDoS attacks.

default

Enabling public network access to cloud resources can affect an organization’s ability to protect its data or internal operations from data theft or disruption.

Depending on the component, inbound access from the Internet can be enabled via:

- a boolean value that explicitly allows access to the public network.
- the assignment of a public IP address.
- database firewall rules that allow public IP ranges.

Deciding to allow public access may happen for various reasons such as for quick maintenance, time saving, or by accident.

This decision increases the likelihood of attacks on the organization, such as:

- data breaches.
- intrusions into the infrastructure to permanently steal from it.
- and various malicious traffic, such as DDoS attacks.

Ask Yourself Whether

This cloud resource:

- **should be publicly accessible to any Internet user.**
- **requires inbound traffic from the Internet to function properly.**

There is a risk if you answered no to any of those questions.

Recommended Secure Coding Practices

Avoid publishing cloud services on the Internet unless they are intended to be publicly accessible, such as customer portals or e-commerce sites.

Use private networks (and associated private IP addresses) and VPC peering or other secure communication tunnels to communicate with other cloud components.

The goal is to prevent the component from intercepting traffic coming in via the public IP address. If the cloud resource does not support the absence of a public IP address, assign a public IP address to it, but do not create listeners for the public IP address.

Sensitive Code Example

For aws-cdk-lib.aws_ec2.Instance and similar constructs:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.Instance(this, "example", {
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    vpcSubnets: {subnetType: ec2.SubnetType.PUBLIC} // Sensitive
})
```

For aws-cdk-lib.aws_ec2.CfnInstance:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnInstance(this, "example", {
    instanceType: "t2.micro",
    imageId: "ami-0ea0f26a6d50850c5",
    networkInterfaces: [
        {
            deviceIndex: "0",
            associatePublicIpAddress: true, // Sensitive
            deleteOnTermination: true,
            subnetId: vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC}).subnetIds[0]
        }
    ]
})
```

For aws-cdk-lib.aws_dms.CfnReplicationInstance:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new dms.CfnReplicationInstance(
    this, "example", {
    replicationInstanceClass: "dms.t2.micro",
    allocatedStorage: 5,
    publiclyAccessible: true, // Sensitive
    replicationSubnetGroupIdentifier: subnetGroup.replicationSubnetGroupIdentifier,
    vpcSecurityGroupIds: [vpc.vpcDefaultSecurityGroup]
})
```

For aws-cdk-lib.aws_rds.CfnDBInstance:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const rdsSubnetGroupPublic = new rds.CfnDBSubnetGroup(this, "publicSubnet", {
    dbSubnetGroupDescription: "Subnets",
    dbSubnetGroupName: "publicSn",
    subnetIds: vpc.selectSubnets({
        subnetType: ec2.SubnetType.PUBLIC
    }).subnetIds
})
```

```
new rds.CfnDBInstance(this, "example", {
    engine: "postgres",
    masterUsername: "foobar",
    masterUserPassword: "12345678",
    dbInstanceClass: "db.r5.large",
    allocatedStorage: "200",
    iops: 1000,
    dbSubnetGroupName: rdsSubnetGroupPublic.ref,
    publiclyAccessible: true, // Sensitive
    vpcSecurityGroups: [sg.securityGroupId]
})
```

Compliant Solution

For **aws-cdk-lib.aws_ec2.Instance** and similar constructs:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.Instance(
    this,
    "example", {
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    vpcSubnets: {subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS}
})
```

For **aws-cdk-lib.aws_ec2.CfnInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnInstance(this, "example", {
    instanceType: "t2.micro",
    imageId: "ami-0ea0f26a6d50850c5",
    networkInterfaces: [
        {
            deviceIndex: "0",
            associatePublicIpAddress: false,
            deleteOnTermination: true,
            subnetId: vpc.selectSubnets({subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS}).subnetIds[0]
        }
    ]
})
```

For **aws-cdk-lib.aws_dms.CfnReplicationInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new dms.CfnReplicationInstance(
    this, "example", {
    replicationInstanceClass: "dms.t2.micro",
    allocatedStorage: 5,
    publiclyAccessible: false,
    replicationSubnetGroupIdentifier: subnetGroup.replicationSubnetGroupIdentifier,
    vpcSecurityGroupIds: [vpc.vpcDefaultSecurityGroup]
})
```

For **aws-cdk-lib.aws_rds.CfnDBInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const rdsSubnetGroupPrivate = new rds.CfnDBSubnetGroup(this, "example",{
    dbSubnetGroupDescription: "Subnets",
    dbSubnetGroupName: "privateSn",
    subnetIds: vpc.selectSubnets({
        subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS
    }).subnetIds
})

new rds.CfnDBInstance(this, "example", {
    engine: "postgres",
    masterUsername: "foobar",
    masterUserPassword: "12345678",
    dbInstanceClass: "db.r5.large",
    allocatedStorage: "200",
    iops: 1000,
    dbSubnetGroupName: rdsSubnetGroupPrivate.ref,
    publiclyAccessible: false,
    vpcSecurityGroups: [sg.securityGroupId]
})
```

See

- **AWS Documentation** - **Amazon EC2 instance IP addressing**
- **AWS Documentation** - **Public and private replication instances**
- **AWS Documentation** - **VPC Peering**
- **CWE** - **CWE-284 - Improper Access Control**
- **CWE** - **CWE-668 - Exposure of Resource to Wrong Sphere**
- **STIG Viewer** - **Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.**

assess_the_problem

Ask Yourself Whether

This cloud resource:

- **should be publicly accessible to any Internet user.**
- **requires inbound traffic from the Internet to function properly.**

There is a risk if you answered no to any of those questions.

Sensitive Code Example

For aws-cdk-lib.aws_ec2.Instance and similar constructs:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.Instance(this, "example", {
    instanceType: nanoT2,
    machineImage: ec2.MachineImage.latestAmazonLinux(),
    vpc: vpc,
    vpcSubnets: {subnetType: ec2.SubnetType.PUBLIC} // Sensitive
})
```

For aws-cdk-lib.aws_ec2.CfnInstance:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new ec2.CfnInstance(this, "example", {
    instanceType: "t2.micro",
    imageId: "ami-0ea0f26a6d50850c5",
    networkInterfaces: [
        {
            deviceIndex: "0",
            associatePublicIpAddress: true, // Sensitive
            deleteOnTermination: true,
            subnetId: vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC}).subnetIds[0]
        }
    ]
})
```

For aws-cdk-lib.aws_dms.CfnReplicationInstance:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

new dms.CfnReplicationInstance(
    this, "example", {
    replicationInstanceClass: "dms.t2.micro",
    allocatedStorage: 5,
    publiclyAccessible: true, // Sensitive
    replicationSubnetGroupIdentifier: subnetGroup.replicationSubnetGroupIdentifier,
    vpcSecurityGroupIds: [vpc.vpcDefaultSecurityGroup]
})
```

For **aws-cdk-lib.aws_rds.CfnDBInstance**:

```
import {aws_ec2 as ec2} from 'aws-cdk-lib'

const rdsSubnetGroupPublic = new rds.CfnDBSubnetGroup(this, "publicSubnet", {
    dbSubnetGroupDescription: "Subnets",
    dbSubnetGroupName: "publicSn",
    subnetIds: vpc.selectSubnets({
        subnetType: ec2.SubnetType.PUBLIC
    }).subnetIds
})

new rds.CfnDBInstance(this, "example", {
    engine: "postgres",
    masterUsername: "foobar",
    masterUserPassword: "12345678",
    dbInstanceClass: "db.r5.large",
    allocatedStorage: "200",
    iops: 1000,
    dbSubnetGroupName: rdsSubnetGroupPublic.ref,
    publiclyAccessible: true, // Sensitive
    vpcSecurityGroups: [sg.securityGroupId]
})
```

Using unencrypted SQS queues is security-sensitive

Clave: javascript:S6330

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Amazon Simple Queue Service (SQS) is a managed message queuing service for application-to-application (A2A) communication. Amazon SQS can store messages encrypted as soon as they are received. In the case that adversaries gain physical access to the storage medium or otherwise leak a message from the file system, for example through a vulnerability in the service, they are not able to access the data.

Ask Yourself Whether

- The queue contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It's recommended to encrypt SQS queues that contain sensitive information. Encryption and decryption are handled transparently by SQS, so no further modifications to the application are necessary.

Sensitive Code Example

For `aws-cdk-lib.aws-sqs.Queue`

```
import { Queue } from 'aws-cdk-lib/aws-sqs';

new Queue(this, 'example', {
    encryption: QueueEncryption.UNENCRYPTED // Sensitive
});
```

For `aws-cdk-lib.aws-sqs.CfnQueue`

```
import { CfnQueue } from 'aws-cdk-lib/aws-sqs';

new CfnQueue(this, 'example', {
    sqsManagedSseEnabled: false // Sensitive
});
```

Compliant Solution

For `aws-cdk-lib.aws-sqs.Queue`

```
import { Queue } from 'aws-cdk-lib/aws-sqs';

new Queue(this, 'example', {
    encryption: QueueEncryption.SQS_MANAGED
});
```

For `aws-cdk-lib.aws-sqs.CfnQueue`

```
import { CfnQueue } from 'aws-cdk-lib/aws-sqs';

new CfnQueue(this, 'example', {
    sqsManagedSseEnabled: true
});
```

See

- **AWS Documentation -** Encryption at rest
- CWE **- CWE-311 - Missing Encryption of Sensitive Data**
- STIG Viewer **- Application Security and Development: V-222588 -** The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

how_to_fix

Recommended Secure Coding Practices

It’s recommended to encrypt SQS queues that contain sensitive information. Encryption and decryption are handled transparently by SQS, so no further modifications to the application are necessary.

Compliant Solution

For `aws-cdk-lib.aws-sqs.Queue`

```
import { Queue } from 'aws-cdk-lib/aws-sqs';

new Queue(this, 'example', {
    encryption: QueueEncryption.SQS_MANAGED
});
```

For `aws-cdk-lib.aws-sqs.CfnQueue`

```
import { CfnQueue } from 'aws-cdk-lib/aws-sqs';
```

```
new CfnQueue(this, 'example', {
    sqsManagedSseEnabled: true
});
```

See

- AWS Documentation - Encryption at rest
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- STIG Viewer - Application Security and Development: V-222588 - The application must implement approved cryptographic mechanisms to prevent unauthorized modification of information at rest.

root_cause

Amazon Simple Queue Service (SQS) is a managed message queuing service for application-to-application (A2A) communication. Amazon SQS can store messages encrypted as soon as they are received. In the case that adversaries gain physical access to the storage medium or otherwise leak a message from the file system, for example through a vulnerability in the service, they are not able to access the data.

assess_the_problem

Ask Yourself Whether

- The queue contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For `aws-cdk-lib.aws-sqs.Queue`

```
import { Queue } from 'aws-cdk-lib/aws-sqs';

new Queue(this, 'example', {
    encryption: QueueEncryption.UNENCRYPTED // Sensitive
});
```

For `aws-cdk-lib.aws-sqs.CfnQueue`

```
import { CfnQueue } from 'aws-cdk-lib/aws-sqs';

new CfnQueue(this, 'example', {
    sqsManagedSseEnabled: false // Sensitive
});
```

Using unencrypted EFS file systems is security-sensitive

Clave: javascript:S6332

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

default

Amazon Elastic File System (EFS) is a serverless file system that does not require provisioning or managing storage. Stored files can be automatically encrypted by the service. In the case that adversaries gain physical access to the storage medium or otherwise leak a message they are not able to access the data.

Ask Yourself Whether

- The file system contains sensitive data that could cause harm when leaked.
- There are compliance requirements for the service to store data encrypted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to encrypt EFS file systems that contain sensitive information. Encryption and decryption are handled transparently by EFS, so no further modifications to the application are necessary.

Sensitive Code Example

For `aws_cdk.aws_efs.FileSystem`

```
import { FileSystem } from 'aws-cdk-lib/aws-efs';

new FileSystem(this, 'unencrypted-explicit', {
    vpc: new Vpc(this, 'VPC'),
    encrypted: false // Sensitive
});
```

For `aws_cdk.aws_efs.CfnFileSystem`

```
import { CfnFileSystem } from 'aws-cdk-lib/aws-efs';

new CfnFileSystem(this, 'unencrypted-implicit-cfn', {
}); // Sensitive as encryption is disabled by default
```

Compliant Solution

For `aws_cdk.aws_efs.FileSystem`

```
import { FileSystem } from 'aws-cdk-lib/aws-efs';

new FileSystem(this, 'encrypted-explicit', {
    vpc: new Vpc(this, 'VPC'),
    encrypted: true
});
```

For `aws_cdk.aws_efs.CfnFileSystem`

```
import { CfnFileSystem } from 'aws-cdk-lib/aws-efs';

new CfnFileSystem(this, 'encrypted-explicit-cfn', {
    encrypted: true
});
```

See

- AWS Documentation - Data encryption in Amazon EFS
- CWE - CWE-311 - Missing Encryption of Sensitive Data

how_to_fix

Recommended Secure Coding Practices

It’s recommended to encrypt EFS file systems that contain sensitive information. Encryption and decryption are handled transparently by EFS, so no further modifications to the application are necessary.

Compliant Solution

For `aws_cdk.aws_efs.FileSystem`

```
import { FileSystem } from 'aws-cdk-lib/aws-efs';

new FileSystem(this, 'encrypted-explicit', {
    vpc: new Vpc(this, 'VPC'),
    encrypted: true
});
```

For `aws_cdk.aws_efs.CfnFileSystem`

```
import { CfnFileSystem } from 'aws-cdk-lib/aws-efs';

new CfnFileSystem(this, 'encrypted-explicit-cfn', {
    encrypted: true
});
```

See

- **AWS Documentation - Data encryption in Amazon EFS**
- **CWE - CWE-311 - Missing Encryption of Sensitive Data**

assess_the_problem

Ask Yourself Whether

- **The file system contains sensitive data that could cause harm when leaked.**
- **There are compliance requirements for the service to store data encrypted.**

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For `aws_cdk.aws_efs.FileSystem`

```
import { FileSystem } from 'aws-cdk-lib/aws-efs';

new FileSystem(this, 'unencrypted-explicit', {
    vpc: new Vpc(this, 'VPC'),
    encrypted: false // Sensitive
});
```

For `aws_cdk.aws_efs.CfnFileSystem`

```
import { CfnFileSystem } from 'aws-cdk-lib/aws-efs';

new CfnFileSystem(this, 'unencrypted-implicit-cfn', {
}); // Sensitive as encryption is disabled by default
```

root_cause

Amazon Elastic File System (EFS) is a serverless file system that does not require provisioning or managing storage. Stored files can be automatically encrypted by the service. In the case that adversaries gain physical access to the storage medium or otherwise leak a message they are not able to access the data.

Creating public APIs is security-sensitive

Clave: javascript:S6333

Severidad: BLOCKER

Estado: ✅ **Activa**

Descripción: No disponible

how_to_fix

Recommended Secure Coding Practices

In general, prefer limiting API access to a specific set of people or entities.

AWS provides multiple methods to do so:

- `AWS_IAM`, **to use standard AWS IAM roles and policies.**
- `COGNITO_USER_POOLS`, **to use customizable OpenID Connect (OIDC) identity providers (IdP).**
- `CUSTOM`, **to use an AWS-independant OIDC provider, glued to the infrastructure with a Lambda authorizer.**

Compliant Solution

For aws-cdk-lib.aws_apigateway.Resource:

```
import {aws_apigateway as apigateway} from "aws-cdk-lib"

const resource = api.root.addResource("example",{
    defaultMethodOptions:{
        authorizationType: apigateway.AuthorizationType.IAM
    }
})
resource.addMethod(
    "POST",
    new apigateway.HttpIntegration("https://example.org"),
    {
        authorizationType: apigateway.AuthorizationType.IAM
    }
)
resource.addMethod(  // authorizationType is inherited from the Resource's configured defaultMethodOptions
    "GET"
)
```

For aws-cdk-lib.aws_apigatewayv2.CfnRoute:

```
import {aws_apigatewayv2 as apigateway} from "aws-cdk-lib"

new apigateway.CfnRoute(this, "auth", {
    apiId: api.ref,
    routeKey: "POST /auth",
    authorizationType: "AWS_IAM",
    target: exampleIntegration
})
```

See

- **AWS Documentation - Controlling and managing access to a REST API in API Gateway**
- **CWE - CWE-284 - Improper Access Control**
- **STIG Viewer - Application Security and Development: V-222620 - Application web servers must be on a separate network segment from the application and database servers.**

root_cause

Creating APIs without authentication unnecessarily increases the attack surface on the target infrastructure.

Unless another authentication method is used, attackers have the opportunity to attempt attacks against the underlying API.
This means attacks both on the functionality provided by the API and its infrastructure.

assess_the_problem

Ask Yourself Whether

- The underlying API exposes all of its contents to any anonymous Internet user.

There is a risk if you answered yes to this question.

Sensitive Code Example

For aws-cdk-lib.aws_apigateway.Resource:

```
import {aws_apigateway as apigateway} from "aws-cdk-lib"

const resource = api.root.addResource("example")
resource.addMethod(
    "GET",
    new apigateway.HttpIntegration("https://example.org"),
    {
        authorizationType: apigateway.AuthorizationType.NONE // Sensitive
    }
)
```

For aws-cdk-lib.aws_apigatewayv2.CfnRoute:

```
import {aws_apigatewayv2 as apigateway} from "aws-cdk-lib"

new apigateway.CfnRoute(this, "no-auth", {
    apiId: api.ref,
    routeKey: "GET /no-auth",
    authorizationType: "NONE", // Sensitive
    target: exampleIntegration
})
```

default

Creating APIs without authentication unnecessarily increases the attack surface on the target infrastructure.

Unless another authentication method is used, attackers have the opportunity to attempt attacks against the underlying API.
This means attacks both on the functionality provided by the API and its infrastructure.

Ask Yourself Whether

- The underlying API exposes all of its contents to any anonymous Internet user.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

In general, prefer limiting API access to a specific set of people or entities.

AWS provides multiple methods to do so:

- `AWS_IAM`, to use standard AWS IAM roles and policies.
- `COGNITO_USER_POOLS`, to use customizable OpenID Connect (OIDC) identity providers (IdP).
- `CUSTOM`, to use an AWS-independant OIDC provider, glued to the infrastructure with a Lambda authorizer.

Sensitive Code Example

For **aws-cdk-lib.aws_apigateway.Resource**:

```
import {aws_apigateway as apigateway} from "aws-cdk-lib"

const resource = api.root.addResource("example")
resource.addMethod(
    "GET",
    new apigateway.HttpIntegration("https://example.org"),
    {
        authorizationType: apigateway.AuthorizationType.NONE // Sensitive
    }
)
```

For **aws-cdk-lib.aws_apigatewayv2.CfnRoute**:

```
import {aws_apigatewayv2 as apigateway} from "aws-cdk-lib"

new apigateway.CfnRoute(this, "no-auth", {
    apiId: api.ref,
    routeKey: "GET /no-auth",
    authorizationType: "NONE", // Sensitive
    target: exampleIntegration
})
```

Compliant Solution

For **aws-cdk-lib.aws_apigateway.Resource**:

```
import {aws_apigateway as apigateway} from "aws-cdk-lib"

const resource = api.root.addResource("example",{
    defaultMethodOptions:{
        authorizationType: apigateway.AuthorizationType.IAM
    }
})
resource.addMethod(
    "POST",
    new apigateway.HttpIntegration("https://example.org"),
    {
        authorizationType: apigateway.AuthorizationType.IAM
    }
)
resource.addMethod(  // authorizationType is inherited from the Resource's configured defaultMethodOptions
    "GET"
)
```

For **aws-cdk-lib.aws_apigatewayv2.CfnRoute**:

```
import {aws_apigatewayv2 as apigateway} from "aws-cdk-lib"

new apigateway.CfnRoute(this, "auth", {
    apiId: api.ref,
    routeKey: "POST /auth",
    authorizationType: "AWS_IAM",
    target: exampleIntegration
})
```

See

- **AWS Documentation** - **Controlling and managing access to a REST API in API Gateway**
- **CWE** - **CWE-284 - Improper Access Control**
- **STIG Viewer** - **Application Security and Development: V-222620** - **Application web servers must be on a separate network segment from the application and database servers.**

Regular expressions with the global flag should be used with caution

Clave: javascript:S6351

Severidad: MAJOR

Estado: ☑ Activa

Descripción: No disponible

root_cause

Regular expressions in JavaScript can have a global flag (`/g`) that enables global searching and matching. While this flag can be useful in certain scenarios, it should be used with caution. When a regular expression has the global flag enabled, it remembers the position of the last match and continues searching for subsequent matches from that position. This behavior can lead to unexpected results if you're not careful and be a source of bugs that are tricky to debug.

The global flag introduces shared state within the regular expression object. This means that if you use the same regular expression object across multiple operations or functions, it maintains its internal state, such as the last match position.

```
const regex = /\d{4}-\d{2}-\d{2}/g;
regex.test('2020-08-06');
regex.test('2019-10-10'); // Noncompliant: the regex will return "false" despite the date being well-formed
```

You should not use the global flag if you intend to use the same regular expression across multiple operations.

```
const regex = /\d{4}-\d{2}-\d{2}/;
regex.test('2020-08-06');
regex.test('2019-10-10');
```

Incorrect usage of global regular expressions can result in infinite loops. For example, if you use a different instance of the same regular expression in a `while`, it can continuously match the same substring, causing an infinite loop.

```
const input = 'foodie fooled football';
while ((result = /foo*/g.exec(input)) !== null) { // Noncompliant: a regex is defined at each iteration causing
  /* ... */
}
```

To avoid an infinite loop, you should create the regular expression with the global flag only once, assign it to a variable, and use the same variable in the loop.

```
const regex = /foo*/g;
const input = 'foodie fooled football';
while ((result = regex.exec(input)) !== null) {
  /* ... */
}
```

Mixing the global flag (`g`) and the sticky flag (`y`) can have different effects on how regular expressions are matched and the behavior of certain methods. The `test()` method ignores the global flag and behaves as if only the sticky flag is set.

```
const regex = /abc/gy; // Noncompliant: a regex enabling both sticky and global flags ignores the global flag
regex.test(/* ... */);
```

Therefore, sou should remove the redundant global flag from the regular expression and only enable the sticky flag.

```
const regex = /abc/y;
regex.test(/* ... */);
```

Overall, this rule raises an issue when:

- a regular expression is tested against different inputs with `RegExp.prototype.test()` or `RegExp.prototype.exec()`
- a regular expression is defined within a loop condition while used with `RegExp.prototype.exec()`
- a regular expression turns on both global `g` and sticky `y` flags

resources

Documentation

- MDN web docs - Regular expression flags
- MDN web docs - `RegExp.prototype.exec()`
- MDN web docs - `RegExp.prototype.test()`

Regular expression quantifiers and character classes should be used concisely

Clave: javascript:S6353

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

A regular expression is a sequence of characters that specifies a match pattern in text. Among the most important concepts are:

- Character classes: defines a set of characters, any one of which can occur in an input string for a match to succeed.
- Quantifiers: used to specify how many instances of a character, group, or character class must be present in the input for a match.
- Wildcard (`.`): matches all characters except line terminators (also matches them if the `s` flag is set).

Many of these features include shortcuts of widely used expressions, so there is more than one way to construct a regular expression to achieve the same results. For example, to match a two-digit number, one could write `[0-9]{2,2}` or `\d{2}`. The latter is not only shorter but easier to read and thus to maintain.

This rule recommends replacing some quantifiers and character classes with more concise equivalents:

- `\d` for `[0-9]` and `\D` for `[^0-9]`
- `\w` for `[A-Za-z0-9_]` and `\W` for `[^A-Za-z0-9_]`
- `.` for character classes matching everything (e.g. `[\w\W]`, `[\d\D]`, or `[\s\S]` with `s` flag)
- `x?` for `x{0,1}`, `x*` for `x{0,}`, `x+` for `x{1,}`, `x{N}` for `x{N,N}`

```
/a{1,}/;        // Noncompliant, '{1,}' quantifier is the same as '+'
/[A-Za-z0-9_]/; // Noncompliant, '\w' is equivalent
```

Use the more concise version to make the regex expression more readable.

```
/a+/;
/\w/;
```

Hard-coded secrets are security-sensitive

Clave: javascript:S6418

Severidad: BLOCKER

Estado: ✅ Activa

Descripción: No disponible

assess_the_problem

Ask Yourself Whether

- The secret allows access to a sensitive component like a database, a file storage, an API, or a service.

- The secret is used in a production environment.
- Application re-distribution is required before updating the secret.

There would be a risk if you answered yes to any of those questions.

Sensitive Code Example

```
const API_KEY = "1234567890abcdef"  // Hard-coded secret (bad practice)

const response = await fetch("https://api.my-service/v1/users", {
  headers: {
    Authorization: `Bearer ${API_KEY}`,
  },
});
```

how_to_fix

Recommended Secure Coding Practices

- Store the secret in a configuration file that is not pushed to the code repository.
- Use your cloud provider’s service for managing secrets.
- If a secret has been disclosed through the source code: revoke it and create a new one.

Compliant Solution

```
const API_KEY = process.env.API_KEY;

const response = await fetch("https://api.my-service/v1/users", {
  headers: {
    Authorization: `Bearer ${API_KEY}`,
  },
});
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials
- MSC - MSC03-J - Never hard code sensitive information

root_cause

Because it is easy to extract strings from an application source code or binary, secrets should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2022-25510
- CVE-2021-42635

Secrets should be stored outside of the source code in a configuration file or a management service for secrets.

This rule detects variables/fields having a name matching a list of words (secret, token, credential, auth, api[_.-]?key) being assigned a pseudorandom hard-coded value. The pseudorandomness of the hard-coded value is based on its entropy and the probability to be human-readable. The randomness sensibility can be adjusted if needed. Lower values will detect less random values, raising potentially more false positives.

default

Because it is easy to extract strings from an application source code or binary, secrets should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2022-25510
- CVE-2021-42635

Secrets should be stored outside of the source code in a configuration file or a management service for secrets.

This rule detects variables/fields having a name matching a list of words (secret, token, credential, auth, api[_.-]?key) being assigned a pseudorandom hard-coded value. The pseudorandomness of the hard-coded value is based on its entropy and the probability to be human-readable. The randomness sensibility can be adjusted if needed. Lower values will detect less random values, raising potentially more false positives.

Ask Yourself Whether

- The secret allows access to a sensitive component like a database, a file storage, an API, or a service.
- The secret is used in a production environment.
- Application re-distribution is required before updating the secret.

There would be a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Store the secret in a configuration file that is not pushed to the code repository.
- Use your cloud provider’s service for managing secrets.
- If a secret has been disclosed through the source code: revoke it and create a new one.

Sensitive Code Example

```
const API_KEY = "1234567890abcdef"  // Hard-coded secret (bad practice)

const response = await fetch("https://api.my-service/v1/users", {
  headers: {
    Authorization: `Bearer ${API_KEY}`,
  },
});
```

Compliant Solution

```
const API_KEY = process.env.API_KEY;

const response = await fetch("https://api.my-service/v1/users", {
  headers: {
    Authorization: `Bearer ${API_KEY}`,
  },
});
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials
- MSC - MSC03-J - Never hard code sensitive information

React Hooks should be properly called

Clave: javascript:S6440

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

React relies on the order in which Hooks are called to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls. This means React Hooks should be called in the same order each time a component renders and should not be called inside loops, conditions, or nested functions.

Additionally, this rule ensures that the Hooks are called only from React function components or custom Hooks.

```
function Profile() {
  const [ordersCount, setOrdersCount] = useState(0);
  if (ordersCount !== 0) {
    useEffect(function() { // Noncompliant: Hook is called conditionally
      localStorage.setItem('ordersData', ordersCount);
    });
  }

  return <div>{ getName() }</div>
}

function getName() {
  const [name] = useState('John'); // Noncompliant: Hook is called from a JavaScript function, not a React comp
  return name;
}
```

Instead, always use Hooks at the top of your React function, before any early returns.

```
function Profile() {
  const [ordersCount, setOrdersCount] = useState(0);
  useEffect(function() {
    if (ordersCount !== 0) {
      localStorage.setItem('ordersData', ordersCount);
    }
  });

  const [name] = useState('John');
  return <div>{ name }</div>
}
```

resources

Documentation

- **React Documentation - Breaking Rules of Hooks**
- **React Documentation - Rules of Hooks**

Unused methods of React components should be removed

Clave: javascript:S6441

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Methods that are never executed are dead code and should be removed. Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced.

When using React class components, all non-React lifecycle methods should be called within the scope of the component. If a method is only called from outside the class, consider using `props` to interact with the component and re-render if needed, as React encourages data-driven components.

```
class Profile extends React.Component {
  render(props) {
    return <h1>{ props.name }</h1>;
  }

  getDefaultName() { // Noncompliant: this method is never used and is a dead code
    return 'John Smith';
  }
}
```

To fix the issue, remove the dead code or call the method from within the component scope.

```
class Profile extends React.Component {
  render(props) {
    return <h1>{ props.name || getDefaultName() }</h1>;
  }

  getDefaultName() {
    return 'John Smith';
  }
}
```

resources

Documentation

- React Documentation - Components and Props
- React Documentation - Adding Lifecycle Methods to a Class

React's useState hook should not be used directly in the render function or body of a component

Clave: javascript:S6442

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

React's `useState` hook setter function should not be called directly in the body of a component, as it would produce an infinite render loop. A re-rendering occurs whenever the state of a component changes. Since a hook setter function changes the component's state, it also triggers re-rendering.

The loop "state updates → triggers re-render → state updates → triggers re-render → …" will continue indefinitely.

```
import { useState } from "react";

function ShowLanguage() {
    const [language, setLanguage] = useState("fr-FR");

    setLanguage(navigator.language); // Noncompliant: causes an infinite loop

    return (
      <section>
        <h1>Your language is {language}!</h1>
        <button onClick={() => setLanguage("fr-FR")}>Je préfère le français</button>
      </section>
    );
}
```

Instead, the setter function should be called from an event handler.

```
import { useState } from "react";

function ShowLanguage() {
    const [language, setLanguage] = useState(navigator.language);

    return (
      <section>
        <h1>Your language is {language}!</h1>
        <button onClick={() => setLanguage("fr-FR")}>Je préfère le Français</button>
      </section>
    );
}
```

resources

Documentation

- React Documentation - React Hooks
- React Documentation - useState - API reference
- React Documentation - useState - Troubleshooting

React state setter function should not be called with its matching state variable

Clave: javascript:S6443

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - State: A Component’s Memory
- React Documentation - useState
- MDN web docs - Object.is()

root_cause

React components have built-in `state` data. This data is used to store component property values. When `state` changes, the component is re-rendered. React provides the `useState` hook to manage the `state`. `useState` returns a state variable retaining the data and a state setter function to update its value.

React will skip re-rendering the component and its children if the new value you provide is identical to the current state, as determined by an `Object.is` comparison. When the setter function is called with the state variable as a parameter, that comparison will always be `true`, and the component will never be re-rendered. This can happen by mistake when attempting to reset a default value or invert a boolean, among others.

This rule raises an issue when calling the setter function with the state variable provided by the same `useState` React hook.

```
import { useState } from "react";

function ShowLanguage() {
    const [language, setLanguage] = useState("fr-FR");
    return (
      <section>
        <h1>Your language is {language}!</h1>
        <button onClick={() => setLanguage(navigator.language)}>Detect language</button>
        <button onClick={() => setLanguage(language)}>Je préfère le français</button>{/* Non compliant: This bu
      </section>
    );
};
```

Instead, you should call the setter with any parameter different from the state variable.

```
import { useState } from "react";

function ShowLanguage() {
    const [language, setLanguage] = useState("fr-FR");
    return (
      <section>
        <h1>Your language is {language}!</h1>
        <button onClick={() => setLanguage(navigator.language)}>Detect language</button>
        <button onClick={() => setLanguage("fr-FR")}>Je préfère le français</button>
      </section>
    );
};
```

JSX list components should have a key property

Clave: javascript:S6477

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- **React Documentation - Rendering lists**
- **React Documentation - Recursing On Children**
- **MDN web docs - Crypto: randomUUID() method**
- **Wikipedia - UUID**

Related rules

- **S6479 - JSX list components should not use array indexes as key**
- **S6486 - JSX list components keys should match up between renders**

root_cause

To optimize the rendering of React list components, a unique identifier (UID) is required for each list item. This UID lets React identify the item throughout its lifetime. To provide it, use the key attribute of the list item. When the key attribute is missing, React will default to using the item's index inside the list component. If the element ordering changes, it will cause keys to not match up between renders, recreating the DOM. It can negatively impact performance and may cause issues with the component state.

```
function Blog(props) {
  return (
    <ul>
      {props.posts.map((post) =>
        <li> <!-- Noncompliant: When 'posts' are reordered, React will need to recreate the list DOM -->
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

To fix it, use a string or a number that uniquely identifies the list item. The key must be unique among its siblings, not globally.

If the data comes from a database, database IDs are already unique and are the best option. Otherwise, use a counter or a UUID generator.

Avoid using array indexes since, even if they are unique, the order of the elements may change.

```
function Blog(props) {
  return (
    <ul>
```

```
      {props.posts.map((post) =>
        <li key={post.id}> <!-- Compliant: id will always be the same even if 'posts' order changes -->
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

React components should not be nested

Clave: javascript:S6478

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

- **React Documentation - Elements Of Different Types**
- **React Documentation - Resetting state with a key**

root_cause

React components should not be nested, as their state will be lost on each re-render of their parent component, possibly introducing bugs. This will also impact performance as child components will be recreated unnecessarily.

If the goal is to have the state reset, use a key instead of relying on a parent state.

```
function Component() {
  function NestedComponent() { // Noncompliant: NestedComponent should be moved outside Component
    return <div />;
  }

  return (
    <div>
      <NestedComponent />
    </div>
  );
}

function Component() {
  return (
    <div>
      <OtherComponent footer={ () => <div /> } /> { /* Noncompliant: Component is created inside prop */ }
    </div>
  );
}

class Component extends React.Component {
  render() {
    function UnstableNestedComponent() { // Noncompliant: NestedComponent should be moved outside Component
      return <div />;
    }

    return (
      <div>
        <UnstableNestedComponent />
      </div>
    );
  }
}
```

You should refactor your code to define a component independently, passing props if needed.

```
function OutsideComponent(props) {
  return <div />;
}

function Component() {
  return (
    <div>
      <OutsideComponent />
    </div>
  );
}

function Component() {
  return <OtherComponent footer={ <div /> } />;
}

class Component extends React.Component {
  render() {
    return (
      <div>
        <OtherComponent />
      </div>
    );
  }
}
```

Component creation is allowed inside component props only if prop name starts with `render`. Make sure you are calling the prop in the receiving component and not using it as an element.

```
function OtherComponent(props) {
  return <div>{props.renderFooter()}</div>;
}

function Component() {
  return (
    <div>
      <OtherComponent renderFooter={() => <div />} />
    </div>
  );
}
```

JSX list components should not use array indexes as key

Clave: javascript:S6479

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

To optimize the rendering of React list components, a unique identifier (UID) is required for each list item. This UID lets React identify the item throughout its lifetime. Avoid array indexes since the order of the items may change, which will cause keys to not match up between renders, recreating the DOM. It can negatively impact performance and may cause issues with the component state.

```
function Blog(props) {
  return (
    <ul>
      {props.posts.map((post, index) =>
        <li key={index}> <!-- Noncompliant: When 'posts' are reordered, React will need to recreate the list DC
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

To fix it, use a string or a number that uniquely identifies the list item. The key must be unique among its siblings, not globally.

If the data comes from a database, database IDs are already unique and are the best option. Otherwise, use a counter or a UUID generator.

```
function Blog(props) {
  return (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
}
```

resources

Documentation

- React Documentation - Rendering lists
- React Documentation - Recursing On Children
- MDN web docs - Crypto: randomUUID() method
- Wikipedia - UUID

Related rules

- S6477 - JSX list components should have a key property
- S6486 - JSX list components keys should match up between renders

---

Import variables should not be reassigned

Clave: javascript:S6522

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Assigning a value to an import variable will cause a runtime error and will raise a compilation error in TypeScript.

Named imports

When using named imports, the imported identifier is a *live binding* exported by another module. Live bindings can be updated or reassigned by the exporting module, and the imported value would also change. The importing module cannot reassign it.

```
import { exportedObject } from 'module.js';
exportedObject = 'hello world!';   // Noncompliant: TypeError: Assignment to constant variable.
```

This rule will not raise an issue when a module mutates the imported object. Be aware that all other modules importing the same value will observe the mutated value.

```
import { exportedObject } from 'module.js';
exportedObject.newAttribute = 'hello world!'; // exportedObject now contains newAttribute and can be seen from
```

Namespace and dynamic imports

This rule will raise an issue when modifying members of a *module namespace object*. A module namespace object is a sealed object that describes all exports from a module.

This can be done using

- a namespace import

```
import * as module from 'module.js';
module.newObject = module.exportedObject; // Noncompliant: TypeError: Cannot add property readPath, object is n
```

- the fulfillment value of a dynamic import.

```
import('module.js').then(module => {
  module.newObject = module.exportedObject; // Noncompliant: TypeError: Cannot add property readPath, object is
})
```

Default imports

Default imports are live bindings to the `default` export. As with the other forms of `import` declarations, the importing module cannot reassign it.

```
import module from 'module.js';
module = 'hello world!';   // Noncompliant: TypeError: Assignment to constant variable.
```

However, the object which `default` refers to is not a *live binding* and may still be mutated by importing modules.

```
import moduleDefault from 'module.js';
moduleDefault.newAttribute = 'hello world!'; // module.default now contains newAttribute and can be seen from a
```

resources

Documentation

- MDN web docs - import
- MDN web docs - Module namespace object
- MDN web docs - Sealed Objects

Numbers should not lose precision

Clave: javascript:S6534

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

For large numbers, JavaScript provides the helper function `Number.isSafeInteger()` to test if a number is between the safe limits.

When you need to store a large number, use `BigInt`. `bigint` and `number` primitives can be compared between them as usual (e.g. `>`, `==`), but pay attention that arithmetic operations (`+ * - % **`) between both types raise an error unless they are converted to the same type. Use the `BigInt` and `Number` functions to convert between both types:

```
const myNumber = Number(myBigInt);
const myBigInt = BigInt(myNumber);
```

Be careful converting values back and forth, however, as the precision of a `bigint` value may be lost when it is coerced to a `number` value.

Noncompliant code example

```
const foo = 2312123211345545367  // Noncompliant: will be stored as 2312123211345545000
const bar = BigInt(2312123211345545367);  // Noncompliant: parameter is first parsed as an integer and thus rou
```

Compliant solution

```
const foo = BigInt('2312123211345545367');
const bar = 2312123211345545367n;
```

When in need of more decimal precision, it is recommended to use a dedicated library to ensure that calculation errors are not introduced by rounding.

Noncompliant code example

```
const baz = 0.123456789123456789 // Noncompliant: will be stored as 0.123456789123456780
```

Compliant solution

```
// use a library like decimal.js for storing numbers containing many decimal digits
import { Decimal } from 'decimal.js';
const bar = new Decimal('0.123456789123456789');
```

root_cause

Numbers in JavaScript are stored in double-precision 64-bit binary format IEEE 754. Like any other number encoding occupying a finite number of bits, it is unable to represent all numbers.

The values are stored using 64 bits in the following form:

- 1 bit for the sign (positive or negative)
- 11 bits for the exponent (2^n). $-1022 \leq n \leq 1023$
- 52 bits for the significand (or mantissa)

The actual value of the stored number will be $(-1)^{sign} * (1 + significand) * 2^{exponent}$

Given this structure, there are limits in both magnitude and precision.

Due to the 52 bits used for the significand, any arithmetic in need of more precision than 2^{-52} (provided by `Number.EPSILON`) is subject to rounding.

In terms of magnitude, the largest number the 64 bits of the format can store is $2^{1024} - 1$ (`Number.MAX_VALUE`).

However, because the 52 bits of the significand, only integers between $-(2^{53} - 1)$ (`Number.MIN_SAFE_INTEGER`) and $2^{53} - 1$ (`Number.MAX_SAFE_INTEGER`) can be represented exactly and be properly compared.

```
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2;  // true
```

JavaScript provides the `bigint` primitive to represent values which are too large to be represented by the number primitive. BigInts are created by appending `n` to the end of an integer literal, or by calling the `BigInt()` function (without the new operator), with an integer or a string.

```
const myBigInt = BigInt(Number.MAX_SAFE_INTEGER);
myBigInt + 1n === myBigInt + 2n;  // false
```

resources

Documentation

- MDN web docs - Number encoding
- MDN web docs - BigInt
- Wikipedia - Double-precision floating-point format
- Wikipedia - IEEE 754 Standard

Unnecessary character escapes should be removed

Clave: javascript:S6535

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The \ (backslash) character indicates that the next character should be treated as a literal character rather than as a special character or string delimiter. For instance, it is common to escape single quotes inside a string literal using the single quote delimiter like `'It\'s a beautiful day'`. Escaping is only meaningful for special characters. Escaping non-special characters in strings, template literals, and regular expressions doesn’t affect their value.

```
const regex = /[\[]/;  // Noncompliant: '[' does not need to be escaped when inside a character class '[]'
const octal = '\8';    // Noncompliant: '8' is not valid octal number
const hello = 'Hello, world\!';    // Noncompliant: '!' is not a special character
const path  = `\/${some}\/${dir}`; // Noncompliant: '/' is not a special character
```

Therefore, useless escapes impact code readability and could even denote a bug in the code if the developer left it by mistake or intended to escape another special character instead.

You should check if the escape character was not misplaced. Useless character escapes can safely be removed without changing the original value.

```
const regex = /[[]/;
const octal = '8';
const hello = 'Hello, world!';
const path  = `/${some}/${dir}`;
```

resources

Documentation

- MDN web docs - Escaping in Regular expressions
- W3Schools - JavaScript strings

Optional chaining should be preferred

Clave: javascript:S6582

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Optional chaining allows to safely access nested properties or methods of an object without having to check for the existence of each intermediate property manually. It provides a concise and safe way to access nested properties or methods without having to write complex and error-prone `null`/`undefined` checks.

This rule flags logical operations that can be safely replaced with the `?.` optional chaining operator.

how_to_fix

Replace with `?.` optional chaining the logical expression that checks for `null`/`undefined` before accessing the property of an object, the element of an array, or calling a function.

Noncompliant code example

```
function foo(obj, arr, fn) {
    if (obj && obj.value) {}
    if (arr && arr[0])    {}
    if (fn && fn(42))     {}
}
```

Compliant solution

```
function foo(obj, arr, fn) {
    if (obj?.value) {}
    if (arr?.[0])   {}
    if (fn?.(42))   {}
}
```

resources

Documentation

- MDN web docs - Optional chaining

"RegExp.exec()" should be preferred over "String.match()"

Clave: javascript:S6594

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - `String.prototype.match()`
- MDN web docs - `RegExp.prototype.exec()`

root_cause

`String.match()` behaves the same way as `RegExp.exec()` when the regular expression does not include the global flag `g`. While they work the same, `RegExp.exec()` can be slightly faster than `String.match()`. Therefore, it should be preferred for better performance.

The rule reports an issue on a call to `String.match()` whenever it can be replaced with semantically equivalent `RegExp.exec()`.

```
'foo'.match(/bar/);
```

Rewrite the pattern matching from `string.match(regex)` to `regex.exec(string)`.

```
/bar/.exec('foo');
```

Prototypes of builtin objects should not be modified

Clave: javascript:S6643

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

By default, JavaScript allows you to modify native object prototypes, such as `Array`, `String`, `Object`, and so on. This means you can add new properties or methods to native objects or override existing ones. While this flexibility can be useful in some instances, it can lead to unexpected behavior, bugs, and compatibility issues.

The rule forbids extending or modifying native JavaScript objects or prototypes, as prototypes of builtin objects should not be modified altogether.

```
Object.prototype.universe = 42;
Object.defineProperty(Array.prototype, "size", { value: 0 });
```

resources

Documentation

- MDN web docs - Object prototypes
- MDN web docs - `Object.defineProperty()`

introduction

Prototypes of builtin objects should not be modified.

Renaming import, export, and destructuring assignments should not be to the same name

Clave: javascript:S6650

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - import
- MDN web docs - export
- MDN web docs - Destructuring assignment

root_cause

Renaming imports, exports, or destructuring assignments to the same name is redundant and can be safely removed. You may accidentally end up with such code if you do a refactoring and change the local name in several places.

```
import { foo as foo } from "bar";
export { foo as foo };
let { foo: foo } = bar;
```

Fix your code to remove the unnecessary renaming.

```
import { foo } from "bar";
export { foo };
let { foo } = bar;
```

Use Object.hasOwn static method instead of hasOwnProperty

Clave: javascript:S6653

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

The `Object.hasOwn()` method was introduced in ES2022 as a replacement for the more verbose `Object.prototype.hasOwnProperty.call()`. These methods return `true` if the specified property of an object exists as its *own* property. If the property is only available further down the prototype chain or

does not exist at all - the methods return `false`.

If you are still using the old method - replace it with a simpler and more concise alternative.

You should also avoid calling the `obj.hasOwnProperty()` method directly, without using `Object.prototype` as a source. This can lead to a runtime error if `obj.prototype` is `null` and therefore `obj.hasOwnProperty` is undefined. The static method `Object.hasOwn()` does not depend on the `obj.prototype` and is therefore safe to use in such cases.

```
Object.prototype.hasOwnProperty.call(obj, "propertyName"); // Noncompliant
Object.hasOwnProperty.call(obj, "propertyName"); // Noncompliant
({}).hasOwnProperty.call(obj, "propertyName"); // Noncompliant
```

To fix the code replace `hasOwnProperty()` with `Object.hasOwn()`

```
Object.hasOwn(obj, "propertyName");
```

resources

Documentation

- MDN web docs - Object.hasOwn()
- MDN web docs - Object.prototype.hasOwnProperty()

__proto__ property should not be used

Clave: javascript:S6654

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- MDN web docs - inheritance and the prototype chain
- MDN web docs - __proto__
- MDN web docs - Object.getPrototypeOf
- MDN web docs - Object.setPrototypeOf

root_cause

JavaScript has a prototypal inheritance model. Each object has an internal property that points to another object, called a `prototype`. That prototype object has a prototype of its own, and the whole sequence is called a prototype chain. When accessing a property or a method of an object, if it is not found at the top level, the search continues through the object's prototype and then further down the prototype chain. This feature allows for very powerful dynamic inheritance patterns but can also lead to confusion when compared to the classic inheritance.

To simplify the access to the prototype of an object some browsers introduced the `__proto__` property, which was later deprecated and removed from the language. The current ECMAScript standard includes `Object.getPrototypeOf` and `Object.setPrototypeOf` static methods that should be used instead of the `__proto__` property.

```
let prototype = foo.__proto__;  // Noncompliant: use Object.getPrototypeOf
foo.__proto__ = bar; // Noncompliant: use Object.setPrototypeOf
```

To fix your code replace `__proto__` with calls to `Object.getPrototypeOf` and `Object.setPrototypeOf` static methods.

```
let prototype = Object.getPrototypeOf(foo);
Object.setPrototypeOf(foo, bar);
```

Octal escape sequences should not be used

Clave: javascript:S6657

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

Octal escape sequences in string literals have been deprecated since ECMAScript 5 and should not be used in modern JavaScript code.

Many developers may not have experience with this format and may confuse it with the decimal notation.

```
let message = "Copyright \251"; // Noncompliant
```

The better way to insert special characters is to use Unicode or hexadecimal escape sequences.

```
let message1 = "Copyright \u00A9";  // unicode
let message2 = "Copyright \xA9";    // hexadecimal
```

resources

Documentation

- MDN web docs - String literals
- MDN web docs - SyntaxError: Octal escape sequences are deprecated

---

"children" and "dangerouslySetInnerHTML" should not be used together

Clave: javascript:S6761

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Passing JSX as children
- React Documentation - Dangerously setting the inner HTML

root_cause

React has a special prop called `dangerouslySetInnerHTML` that allows you to assign a raw HTML string to the underlying DOM `innerHTML` property. Changing `innerHTML` will replace the element's child nodes or text content. For this reason, `dangerouslySetInnerHTML` should never be used together with component `children` as they will conflict with each other, both trying to set the inner content of the same element.

```
function MyComponent() {
    return ( // Noncompliant: don't use children and dangerouslySetInnerHTML at the same time
        <div dangerouslySetInnerHTML={{ __html: "HTML" }}>
            Children
        </div>
    );
}
```

To fix the issue leave either the element's `children` or `dangerouslySetInnerHTML`, but not both.

```
function MyComponent() {
    return (
        <div dangerouslySetInnerHTML={{ __html: "HTML" }} />
    );
}
```

User-defined JSX components should use Pascal case

Clave: javascript:S6770

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

User-defined JSX components should use Pascal case because it is a widely accepted convention in the React community. Using Pascal case for component names helps to distinguish them from HTML elements and built-in React components, which are typically written in lowercase. It also improves code readability and makes it easier to differentiate between components and regular HTML tags.

Additionally, adhering to this convention ensures consistency and makes it easier for other developers to understand and work with your code. It is considered a best practice in React development and is recommended by the official React documentation.

```
<MY_COMPONENT />
```

You should rename your component according to Pascal case and all its usages as JSX elements.

```
<MyComponent />
```

resources

Documentation

- React Documentation - Creating and nesting components

Spacing between inline elements should be explicit

Clave: javascript:S6772

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

resources

Documentation

- React Documentation - Writing markup with JSX
- MDN web docs - Spaces in between inline and inline-block elements

root_cause

React JSX differs from the HTML standard in the way it handles newline characters and surrounding whitespace. HTML collapses multiple whitespace characters (including newlines) into a single whitespace, but JSX removes such sequences completely, leaving no space between inline elements separated by the line break. This difference in behavior can be confusing and may result in unintended layout, for example, missing whitespace between the link content and the surrounding text.

To avoid such issues, you should never rely on newline characters in JSX, and explicitly specify whether you want whitespace between inline elements separated by a line break.

```
<div>{/* Noncompliant: ambiguous spacing */}
  Here is some
  <a>space</a>
</div>

<div>{/* Noncompliant: ambiguous spacing */}
  <a>No space</a>
  between these
</div>
```

To fix the issue, either insert an explicit JSX space as a string expression `{' '}`, or insert an empty comment expression `{/* */}` to indicate that the two parts will be joined together with no space between them.

```
<div>
  Here is some{' '}
  <a>space</a>
</div>

<div>
  <a>No space</a>{/*
  */}between these
</div>
```

React components should validate prop types

Clave: javascript:S6774

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

Noncompliant code example

```
import PropTypes from 'prop-types';

function Hello({ firstname, lastname }) {
  return <div>Hello {firstname} {lastname}</div>; // Noncompliant: 'lastname' type is missing
}
Hello.propTypes = {
  firstname: PropTypes.string.isRequired
};

// Using legacy APIs

class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.firstname} {this.props.lastname}</div>; // Noncompliant: 'lastname' type is m
  }
}
Hello.propTypes = {
  firstname: PropTypes.string.isRequired,
};
```

Compliant solution

```
import PropTypes from 'prop-types';

function Hello({ firstname, lastname }) {
  return <div>Hello {firstname} {lastname}</div>;
}
```

```
Hello.propTypes = {
  firstname: PropTypes.string.isRequired,
  lastname: PropTypes.string.isRequired,
};

// Using legacy APIs

class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.firstname} {this.props.lastname}</div>;
  }
}
Hello.propTypes = {
  firstname: PropTypes.string.isRequired,
  lastname: PropTypes.string.isRequired,
};
```

root_cause

In JavaScript, props are typically passed as plain objects, which can lead to errors and confusion when working with components that have specific prop requirements. However, it lacks of type safety and clarity when passing props to components in a codebase.

By defining types for component props, developers can enforce type safety and provide clear documentation for the expected props of a component. This helps catch potential errors at compile-time. It also improves code maintainability by making it easier to understand how components should be used and what props they accept.

resources

Documentation

- React Documentation - static propTypes
- Flow.js Documentation - React

All "defaultProps" should have non-required PropTypes

Clave: javascript:S6775

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

how_to_fix

```
function MyComponent({foo, bar}) {
  return <div>{foo}{bar}</div>;
}

MyComponent.propTypes = {
  foo: React.PropTypes.string.isRequired,
};

MyComponent.defaultProps = {
  foo: "foo", // Noncompliant: foo is a required prop
  bar: "bar", // Noncompliant: bar propType is missing
};
```

To fix the issue, verify that each `defaultProp` has a corresponding `propType` definition and is not marked as `isRequired`.

```
function MyComponent({foo, bar}) {
  return <div>{foo}{bar}</div>;
}

MyComponent.propTypes = {
  foo: React.PropTypes.string,
  bar: React.PropTypes.string,
};

MyComponent.defaultProps = {
  foo: "foo",
  bar: "bar",
};
```

resources

Documentation

- React Documentation - TypeScript with React Components
- React Documentation - Specifying a default value for a prop
- React Documentation - Typechecking With PropTypes

how_to_fix

```
type Props = {
  foo: string,
  bar?: string
}

function MyComponent({foo, bar}: Props) {
  return <div>{foo}{bar}</div>;
}

MyComponent.defaultProps = {
  foo: "foo", // Noncompliant: foo is a required prop
  bar: "bar",
};
```

To fix the issue, verify that each `defaultProp` has a corresponding type definition and is marked as optional.

```
type Props = {
  foo?: string,
  bar?: string
}

function MyComponent({foo, bar}: Props) {
  return <div>{foo}{bar}</div>;
}

MyComponent.defaultProps = {
  foo: "foo",
  bar: "bar",
};
```

root_cause

React Legacy APIs provide a way to define the default values for props and check the prop types at runtime. This rule verifies if a `defaultProps` definition does have a corresponding `propTypes` definition. If it is missing, this could be the result of errors in refactoring or a spelling mistake.

It is also an error if a `defaultProp` has `propType` that is marked as `isRequired`.

Equality operators should not be used in "for" loop termination conditions

Clave: javascript:S888

Severidad: CRITICAL

Estado: ✅ Activa

Descripción: No disponible

root_cause

Testing `for` loop termination using an equality operator (`==` and `!=`) is dangerous, because it could set up an infinite loop. Using a broader relational operator instead casts a wider net, and makes it harder (but not impossible) to accidentally write an infinite loop.

Noncompliant code example

```
for (var i = 1; i != 10; i += 2)  // Noncompliant. Infinite; i goes from 9 straight to 11.
{
  //...
}
```

Compliant solution

```
for (var i = 1; i <= 10; i += 2)  // Compliant
{
  //...
}
```

Exceptions

Equality operators are ignored if the loop counter is not modified within the body of the loop and either:

- starts below the ending value and is incremented by 1 on each iteration.
- starts above the ending value and is decremented by 1 on each iteration.

Equality operators are also ignored when the test is against `null`.

```
for (var i = 0; arr[i] != null; i++) {
  // ...
}

for (var i = 0; (item = arr[i]) != null; i++) {
  // ...
}
```

resources

- CWE - CWE-835 - Loop with Unreachable Exit Condition ('Infinite Loop')

Creating cookies without the "secure" flag is security-sensitive

Clave: javascript:S2092

Severidad: MINOR

Estado: ✅ Activa

Descripción: No disponible

default

When a cookie is protected with the `secure` attribute set to *true* it will not be send by the browser over an unencrypted HTTP request and thus cannot be observed by an unauthorized person during a man-in-the-middle attack.

Ask Yourself Whether

- the cookie is for instance a *session-cookie* not designed to be sent over non-HTTPS communication.
- it's not sure that the website contains mixed content or not (ie HTTPS everywhere or not)

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- It is recommended to use `HTTPs` everywhere so setting the `secure` flag to *true* should be the default behaviour when creating cookies.
- Set the `secure` flag to *true* for session-cookies.

Sensitive Code Example

cookie-session module:

```
let session = cookieSession({
  secure: false,// Sensitive
});  // Sensitive
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    secure: false // Sensitive
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  secure: false // Sensitive
}); // Sensitive
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { secure: false }}); // Sensitive
```

Compliant Solution

cookie-session module:

```
let session = cookieSession({
  secure: true,// Compliant
});  // Compliant
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    secure: true // Compliant
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  secure: true // Compliant
}); // Compliant
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { secure: true }}); // Compliant
```

See

- **OWASP - Top 10 2021 Category A4 - Insecure Design**
- **OWASP - Top 10 2021 Category A5 - Security Misconfiguration**
- **OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure**
- **CWE - CWE-311 - Missing Encryption of Sensitive Data**
- **CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie**
- **CWE - CWE-614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute**
- **STIG Viewer - Application Security and Development: V-222576 - The application must set the secure flag on session cookies.**

how_to_fix

Recommended Secure Coding Practices

- **It is recommended to use `HTTPs` everywhere so setting the `secure` flag to *true* should be the default behaviour when creating cookies.**
- **Set the `secure` flag to *true* for session-cookies.**

Compliant Solution

cookie-session module:

```
let session = cookieSession({
  secure: true,// Compliant
});  // Compliant
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    secure: true // Compliant
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  secure: true // Compliant
}); // Compliant
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { secure: true }}); // Compliant
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie
- CWE - CWE-614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- STIG Viewer - Application Security and Development: V-222576 - The application must set the secure flag on session cookies.

assess_the_problem

Ask Yourself Whether

- the cookie is for instance a *session-cookie* not designed to be sent over non-HTTPS communication.
- it’s not sure that the website contains mixed content or not (ie HTTPS everywhere or not)

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

cookie-session module:

```
let session = cookieSession({
  secure: false,// Sensitive
});  // Sensitive
```

express-session module:

```
const express = require('express');
const session = require('express-session');

let app = express();
app.use(session({
  cookie:
  {
    secure: false // Sensitive
  }
}));
```

cookies module:

```
let cookies = new Cookies(req, res, { keys: keys });

cookies.set('LastVisit', new Date().toISOString(), {
  secure: false // Sensitive
}); // Sensitive
```

csurf module:

```
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const express = require('express');

let csrfProtection = csrf({ cookie: { secure: false }}); // Sensitive
```

root_cause

When a cookie is protected with the `secure` attribute set to *true* it will not be send by the browser over an unencrypted HTTP request and thus cannot be observed by an unauthorized person during a man-in-the-middle attack.

Regular expressions should not contain empty groups

Clave: javascript:S6331

Severidad: MAJOR

Estado: ✅ Activa

Descripción: No disponible

root_cause

There are several reasons to use a group in a regular expression:

- **to change the precedence (e.g. `do(g|or)` will match 'dog' and 'door')**
- **to remember parenthesised part of the match in the case of capturing group**
- **to improve readability**

In any case, having an empty group is most probably a mistake. Either it is a leftover after refactoring and should be removed, or the actual parentheses were intended and were not escaped.

Noncompliant code example

```
const dateRegex = /^(?:0[1-9]|[12][0-9]|3[01])[- /.](?:0[1-9]|1[012])[- /.](?:19|20)\d\d(?:)$/; // Noncompliant
const methodCallRegex = /foo()/;  // Noncompliant, will match only 'foo'
```

Compliant solution

```
const dateRegex = /^(?:0[1-9]|[12][0-9]|3[01])[- /.](?:0[1-9]|1[012])[- /.](?:19|20)\d\d$/;
const methodCallRegex = /foo\(\)/; // OK, matches 'foo()'
```
