centro criptológico nacional

Guía de Desarrollo Seguro
CCN-STIC 481 – Anexo C

Codificación Segura para JavaScript



Noviembre 2026

MINISTERIO
DE DEFENSA

cpage.mpr.gob.es

Catálogo de Publicaciones de la Administración General del Estado
cpage.mpr.gob.es

Edita:

MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA

CCN
centro criptológico nacional

Pº de la Castellana 109, 28046 Madrid
© Centro Criptológico Nacional, 2026
NIPO (edición en línea)

CCN
c=ES, l=MADRID, o=CCN,
ou=SELLO ELECTRONICO,
2.5.4.97=VATES-S2800155J,
serialNumber=S2800155J,
cn=CCN
2026.11.23 11:35:52 +01:00

LIMITACIÓN DE RESPONSABILIDAD
El presente documento se proporciona de acuerdo con los términos en él recogidos, rechazando expresamente cualquier tipo de garantía implícita que se pueda encontrar relacionada. En ningún caso, el Centro Criptológico Nacional puede ser considerado responsable del daño directo, indirecto, fortuito o extraordinario derivado de la utilización de la información y software que se indican incluso cuando se advierta de tal posibilidad.

AVISO LEGAL
Quedan rigurosamente prohibidas, sin la autorización escrita del Centro Criptológico Nacional, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de este documento por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares del mismo mediante alquiler o préstamo públicos.

Esta publicación se distribuye bajo licencia CC BY-NC-ND 4.0 que permite compartir el material en cualquier medio o formato con la condición de reconocer adecuadamente la procedencia «Edita: Ministerio de Defensa. Secretaría General Técnica.» No se puede modificar. No se puede utilizar con fines comerciales.

ÍNDICE

1. INTRODUCCIÓN 1

1.1 OBJETIVO 1
1.2 ALCANCE 1
1.3 ESTÁNDARES METODOLÓGICOS 1
1.3.1 The JavaScript Guide: Web Application Secure Coding Practices 2
1.3.2 Airbnb JavaScript Style Guide 2
1.3.3 Google JavaScript Style Guide 3
1.3.4 MDN Web Docs: Javascript 3
1.4 ESTRUCTURA DE LA GUÍA 4

2. REGLAS DE CODIFICACIÓN SEGURA 6

2.1 ESTRUCTURAS DE CONTROL Y FLUJO 6
2.1.1 [js.flow.1] - Evitar el uso de "eval" y del constructor Function 6
2.1.2 [js.flow.2] - Prohibir el uso de la cláusula "with" 10
2.1.3 [js.flow.3] - No depender del hoisting implícito de variables y funciones 14
2.1.4 [js.flow.4] - Usar siempre bloques explícitos en estructuras de control 18
2.2 GESTIÓN DE MEMORIA 21
2.2.1 [js.mem.1] - Evitar el uso implícito de variables globales 21
2.2.2 [js.mem.2] - No reutilizar referencias a objetos mutables compartidos entre contextos asíncronos sin un control explícito 25
2.2.3 [js.mem.3] - Liberar referencias a objetos que ya no se usan 29
2.3 VALIDACIONES 33
2.3.1 [js.val.1] - No construir código ejecutable a partir de entrada externa sin validación estricta 33
2.3.2 [js.val.2] - Validar y sanear todas las entradas externas antes de su uso 37
2.3.3 [js.val.3] - Evitar el uso de APIs obsoletas o inseguras 41
2.4 GESTIÓN DE ERRORES 45
2.4.1 [js.err.1] - Capturar y gestionar todas las excepciones 45
2.4.2 [js.err.2] - Evitar lanzar valores no-Error como excepciones 49
2.4.3 [js.err.3] - Evitar la ejecución de errores silenciosos 54
2.5 CONCURRENCIA 59
2.5.1 [js.sync.1] - Gestionar errores en promesas mediante try/catch con async/await 59
2.5.2 [js.sync.2] - Proteger el acceso a recursos compartidos en código asíncrono 63
2.5.3 [js.sync.3] - No mezclar callbacks y promesas sin control claro del flujo de ejecución 68
2.5.4 [js.sync.4] - No pasar cadenas de texto a "setTimeout" o "setInterval" 73
2.5.5 [js.sync.5] - Controlar explícitamente el ciclo de vida de setInterval y temporizadores 76

3. ANEXO: METODOLOGÍA DE CÁLCULO DEL SCORE CWSS 81

1. Introducción

1.1 Objetivo

El propósito de este documento es ofrecer un **catálogo exhaustivo de controles técnicos de ciberseguridad**, diseñado para servir como referencia práctica en la implementación de medidas de seguridad efectivas durante el desarrollo de software.

Dicho catálogo se encuentra alineado con las mejores prácticas en materia de desarrollo seguro y tiene como objetivo **proporcionar soluciones técnicas seguras**, validadas y contrastadas frente a las vulnerabilidades más comunes que pueden presentarse en cada ámbito de la implementación.

El documento pretende facilitar la **integración sistemática de la seguridad en el ciclo de vida del software** (S-SDLC), promoviendo un enfoque preventivo, estandarizado y verificable que contribuya a la reducción de riesgos y a la mejora continua.

1.2 Alcance

La correcta comprensión y aplicación de **los principios** contenidos en este documento **son de carácter obligatorio**, ya que su cumplimiento garantiza la alineación con los estándares, políticas internas y regulaciones de seguridad vigentes, así como con las mejores prácticas internacionales en materia de desarrollo seguro.

Se recomienda la lectura y conocimiento **a todos los miembros del equipo de desarrollo**, incluyendo aquellos que participen directa o indirectamente en cualquiera de las fases restantes del S-SDLC, como análisis de requisitos, pruebas, despliegue u operaciones. Esto permitirá fomentar una cultura de seguridad transversal, asegurando que todos los involucrados comprendan la importancia de integrar la seguridad como un elemento esencial a lo largo de todo el ciclo de vida del software.

1.3 Estándares metodológicos

Las **reglas de codificación segura** descritas en este documento **se fundamentan en las recomendaciones y buenas prácticas** definidas por las guías más utilizadas y mejor respaldadas por la industria.

Cada regla hace referencia explícita a la fuente normativa o estándar de seguridad que lo respalda, asegurando su trazabilidad y su alineación con los marcos regulatorios vigentes.

1.3.1 The JavaScript Guide: Web Application Secure Coding Practices

https://github.com/Checkmarx/JS-SCP

Guía de codificación segura para JavaScript orientada al desarrollo de aplicaciones web, publicada por Checkmarx. Cubre los patrones de escritura de código que previenen las vulnerabilidades más comunes en aplicaciones JavaScript del lado del cliente y del servidor. Trata la validación y saneamiento de entradas, la prevención de XSS (Cross-Site Scripting), inyección de código, manejo seguro del DOM, uso correcto de cookies y cabeceras HTTP de seguridad, gestión de tokens de autenticación y autorización, y protección frente a ataques de tipo CSRF. Incluye también directrices sobre el uso seguro de APIs externas, el manejo de datos sensibles en el navegador, la configuración de Content Security Policy (CSP) y la evitación de patrones de código que exponen superficie de ataque. A diferencia de las guías de estilo convencionales, el criterio principal de cada regla no es la legibilidad ni la coherencia, sino la reducción del riesgo de explotación. Está orientada a equipos de desarrollo que trabajan bajo requisitos de seguridad formales o que integran análisis estático de seguridad en su pipeline de CI/CD.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **JS-WASCP-2.1**: Validation.
- **JS-WASCP-3.1.3**: How to prevent.
- **JS-WASCP-14.1**: Dependencies.

1.3.2 Airbnb JavaScript Style Guide

https://github.com/airbnb/javascript

Guía de estilo para JavaScript mantenida por Airbnb y ampliamente adoptada en la industria. Define convenciones de escritura de código con el objetivo de mejorar la legibilidad, coherencia y mantenibilidad en proyectos de equipo. Cubre el uso preferente de `const` y `let` sobre `var`, reglas de formato (indentación, punto y coma, comillas), convenciones de nomenclatura, uso de funciones flecha, destructuring, módulos ES6, clases y programación asíncrona. Incluye también directrices sobre cómo importar módulos, estructurar componentes y evitar patrones considerados propensos a errores. Se distribuye como un conjunto de reglas para ESLint, lo que permite aplicar y verificar automáticamente el cumplimiento de la guía durante el desarrollo. Es uno de los conjuntos de convenciones más extendidos en proyectos JavaScript y TypeScript.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **Airbnb-JS-7**: Functions.
- **Airbnb-JS-13**: Variables.

- **Airbnb-JS-29**: Standard Library.

1.3.3 Google JavaScript Style Guide

https://google.github.io/styleguide/jsguide.html

Guía de estilo para JavaScript publicada y mantenida por Google. Define las convenciones de escritura de código adoptadas internamente en sus proyectos y que han sido liberadas como referencia pública. Cubre el uso obligatorio de `const` y `let`, la prohibición explícita de `var`, reglas de formato (indentación de dos espacios, comillas simples, punto y coma obligatorio), convenciones de nomenclatura para variables, funciones, clases y constantes, y el uso de módulos ES6. Presta especial atención al uso correcto de tipos, la documentación mediante JSDoc, el manejo de promesas y `async/await`, y no usar patrones ambiguos o dependientes del comportamiento implícito del motor. A diferencia de la guía de Airbnb, tiene un enfoque más conservador y explícito, orientado a bases de código grandes y equipos numerosos. Se complementa con la Google Closure Compiler y las anotaciones de tipo que este requiere, aunque su aplicación no está limitada a ese entorno.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **Google-JS-5.1**: Local variable declarations.
- **Google-JS-5.8**: Control structures.
- **Google-JS-4.1**: Braces.

1.3.4 MDN Web Docs: Javascript

https://developer.mozilla.org/en-US/docs/Web/JavaScript

Documentación de referencia mantenida por Mozilla para el lenguaje JavaScript y las APIs web. Cubre desde los fundamentos del lenguaje (tipos, operadores, estructuras de control, funciones, clases, módulos) hasta características avanzadas como closures, prototipos, iteradores, generadores y programación asíncrona con `Promises` y `async/await`. Incluye también la referencia completa de las APIs del navegador: DOM, `Fetch`, Web Storage, Web Workers, y otras interfaces estándar. Cada artículo combina descripción de la especificación, ejemplos de uso y notas de compatibilidad entre navegadores. Es la referencia de facto para el desarrollo web del lado del cliente.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **MDN Web Docs JS-2**: Grammar and types.
- **MDN Web Docs JS-4**: Control flow and error handling.
- **MDN Web Docs JS-18**: Resource management.

1.4 Estructura de la guía

La guía está constituida por una lista de reglas de codificación segura, organizada en las siguientes categorías:

- Estructuras de control y flujo.
- Gestión de memoria.
- Validaciones.
- Gestión de errores.
- Concurrencia.

Los epígrafes de cada categoría representan reglas de la categoría. Cada una de ellas definida por un **identificador** y el **nombre** de la regla.

Debajo de cada regla, se muestra una pequeña tabla, a **modo de cabecera**, que resume las características más relevantes como son:

- **ID**: identificador de la regla.
- **Uso:** obligatorio o recomendado.
- **CWSS Score**: la puntuación asignada mediante evaluación CWSS (0-100).
- **Impacto técnico**: la severidad del daño que podría producir una codificación insegura de la regla (Muy alto, Alto, Medio, Bajo, Muy bajo). Su impacto queda reflejado en la puntuación del CWSS.
- **Dimensión**: la dimensión de seguridad afectada (Integridad, Confidencialidad, Disponibilidad).
- **S-SDLC**: la fase del S-SDLC donde debería tenerse presente la regla de seguridad. De entre todas las fases, los únicos valores afectados son **Design**, **Build** y **Test**.

ID	**cpp.const.1**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Detalle gráfico de todos los campos de la cabecera de cada regla

Después de la cabecera se muestra la siguiente información:

- Una **breve descripción**, que pondrá en antecedentes al usuario sobre el significado de la regla.

- Dos ejemplos compilables de código: una **solución no recomendada** y la **solución segura recomendada**.
- Una descripción extensa y detallada para profundizar en el significado de la regla, y las consecuencias de ignorarla durante el desarrollo.
- Una **lista** con todos los **riesgos de seguridad** en los que se incurriría en caso de ignorar la regla.
- Una **lista de las buenas prácticas** recomendadas para aplicar la regla en las situaciones más comunes junto con ejemplos de malas y buenas implementaciones.
- Los **estándares metodológicos** a los que hace referencia.

En todos los ejemplos de código, las soluciones no recomendadas se mostrarán siempre en **color rojo** para que puedan ser identificarlas rápidamente como implementaciones a evitar. Y las soluciones seguras, o recomendadas, en **color azul**.

2. Reglas de codificación segura

2.1 Estructuras de control y flujo

2.1.1 [js.flow.1] - Evitar el uso de "eval" y del constructor Function

ID	js.flow.1	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones `eval()` y el constructor `Function()` ejecutan cadenas de texto como código, creando vectores de ataque que permiten la **inyección de código malicioso**. Estos mecanismos eliminan las protecciones del intérprete y pueden comprometer completamente la seguridad de la aplicación al permitir ejecución arbitraria de instrucciones.

Codificación insegura

Uso de `eval()` o `Function()` para ejecutar una operación recibida como texto desde la entrada del usuario. Se busca mostrar cómo una funcionalidad simple de calculadora puede convertirse en un punto de ejecución arbitraria de código y pérdida de control del flujo del programa.

```
//-- Módulo para leer desde consola
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
//-- Solicita al usuario una operación matemática
rl.question('Introduce una operación matemática (ej: 2 + 2): ', (input) => {
  try {
    //-- Uso inseguro de Function() con entrada controlada por el usuario
    const fn = new Function(`return (${input})`);
    const result = fn();
    //-- Otra solución igualmente insecura con eval()
    const result2 = eval(input);
  } catch (err) {
    //-- Manejo básico de errores
    console.error('Error al evaluar la expresión:', err.message);
  }
  rl.close();
});
```

Codificación recomendada

Implementación de una calculadora simple sin usar `eval` ni el constructor `Function`, validando la operación y aplicando únicamente un conjunto controlado de operadores permitidos. Se pretende demostrar cómo mantener el control del flujo y evitar ejecución arbitraria.

```
'use strict';

//-- Módulo para leer desde consola
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

//-- Operaciones permitidas definidas explícitamente
const operations = {
  '+': (a, b) => a + b,
  '-': (a, b) => a - b,
  '*': (a, b) => a * b,
  '/': (a, b) => b !== 0 ? a / b : NaN
};

//-- Solicita al usuario una operación en formato controlado
rl.question('Introduce una operación (ej: 2 + 2): ', (input) => {
  try {
    //-- Separar la entrada en partes
    const parts = input.trim().split(' ');

    //-- Validar formato esperado
    if (parts.length !== 3) {
      throw new Error('Formato inválido');
    }
    const a = Number(parts[0]);
    const operator = parts[1];
    const b = Number(parts[2]);

    //-- Validar números y operador permitido
    if (Number.isNaN(a) || Number.isNaN(b) || !operations[operator]) {
      throw new Error('Entrada no válida');
    }
    //-- Ejecutar únicamente operaciones definidas
    const result = operations[operator](a, b);
    console.log('Resultado:', result);
  } catch (err) {
    //-- Manejo controlado de errores
    console.error('Error:', err.message);
  }
```

```
  rl.close();
});
```

2.1.1.1 Descripción

La función `eval` y el constructor `Function` ejecutan cadenas de texto como código en tiempo de ejecución. Esta capacidad introduce vulnerabilidades significativas al permitir la ejecución de código arbitrario que puede no estar bajo el control directo del desarrollador.

Ambas funciones interpretan y ejecutan cualquier contenido que reciban como parámetro, lo que las convierte en **vectores de ataque potenciales** cuando procesan datos no confiables. La naturaleza dinámica de estas funciones impide que los analizadores estáticos de código detecten vulnerabilidades durante el desarrollo.

El uso de estas funciones compromete la **integridad del flujo de ejecución** del programa, ya que introducen puntos donde el comportamiento del código no puede predecirse completamente durante la fase de desarrollo. Esta impredecibilidad afecta tanto la seguridad como la mantenibilidad del código.

Los riesgos asociados incluyen la posibilidad de inyección de código malicioso, acceso no autorizado a variables del ámbito local, y la ejecución de operaciones no previstas por el desarrollador. Estos problemas se amplifican cuando las funciones procesan datos provenientes de fuentes externas o de la entrada del usuario.

La mitigación de estos riesgos requiere evitar completamente el uso de `eval` y el constructor `Function`, optando por alternativas más seguras como **parsers específicos** para datos estructurados, funciones nativas para manipulación de objetos o bibliotecas especializadas que no requieren ejecución dinámica de código. En casos donde la funcionalidad dinámica sea imprescindible, se debe implementar **validación estricta** y sandboxing apropiado.

Riesgos asociados

- Ejecución de código malicioso inyectado por un atacante.
- Pérdida de control sobre el flujo de ejecución de la aplicación.
- Exposición a ataques de inyección de código JavaScript (Cross-Site Scripting, XSS).
- Dificultad para realizar análisis estático de código y pruebas de seguridad.
- Imposibilidad de predecir el comportamiento del sistema ante entradas dinámicas.
- Incumplimiento de las políticas de Content Security Policy (CSP), reduciendo la protección del navegador.
- Dificultad en el mantenimiento y refactorización del código, aumentando la

probabilidad de errores.

- Riesgo de fugas de información sensible por acceso a variables y funciones fuera del alcance previsto.
- Aumento significativo de la superficie de ataque de la aplicación.
- Impacto negativo en el rendimiento de la aplicación debido a la interpretación dinámica del código.

Buenas prácticas

- Evitar el uso de `eval` para ejecutar cadenas dinámicas; utilizar estructuras de control o funciones predefinidas.

```
//-- Construcción y ejecución dinámica de código basada en entrada externa
const operacion = userInput;
const resultado = eval("2 + " + operacion);
```

```
//-- Uso de validación estricta y operaciones controladas
const operacionSegura = Number(userInput);
if (Number.isFinite(operacionSegura)) {
  const resultado = 2 + operacionSegura;
}
```

- No utilizar el constructor `Function` para crear funciones dinámicamente; definir funciones estáticas o usar mapas de funciones.

```
//-- Creación dinámica de función a partir de texto externo
const fn = new Function("a", "b", userCode);
const resultado = fn(1, 2);
```

```
//-- Uso de un objeto mapa con funciones previamente definidas
const operaciones = {
  suma: (a, b) => a + b,
  resta: (a, b) => a - b
};
const resultado = operaciones[userOperacion]?.(1, 2);
```

- Evitar temporizadores con cadenas en `setTimeout` o `setInterval`; pasar funciones en su lugar.

```
//-- Ejecución de cadena evaluada implícitamente por el temporizador
setTimeout("procesar('" + dato + "')", 1000);
```

```
//-- Paso de función como callback sin evaluación dinámica
setTimeout(() => {
```

```
  procesar(dato);
}, 1000);
```

- Validar y transformar datos externos en lugar de interpretarlos como código ejecutable.

```
//-- Interpretación directa de datos JSON manipulados como código
const config = eval("(" + inputConfig + ")");
```

```
//-- Uso de JSON.parse con manejo de errores
let config;
try {
  config = JSON.parse(inputConfig);
} catch (e) {
  config = {};
}
```

2.1.1.2 Estándares metodológicos

- **JS-WASCP-14.2**: Interpreted Code Integrity.
- **JS-WASCP-2.1**: Validation.
- **Airbnb-JS-7**: Functions.

2.1.2 [js.flow.2] - Prohibir el uso de la cláusula "with"

ID	js.flow.2	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La declaración `with` modifica dinámicamente la cadena de ámbitos durante la ejecución, creando comportamientos **no deterministas** que comprometen la predictibilidad del código. Esta alteración del **scope chain** genera ambigüedades en la resolución de variables y dificulta el análisis estático, introduciendo vulnerabilidades de seguridad y errores difíciles de detectar.

Codificación insegura

Uso de `with` para acceder a propiedades de un objeto modificando dinámicamente el scope chain. Se busca mostrar cómo esta práctica puede generar ambigüedad en la resolución de variables, dificultar el mantenimiento y provocar comportamientos inesperados cuando existen variables con el mismo nombre en distintos ámbitos.

```
//-- Definición de una variable global
```

```
let total = 100;

//-- Función que utiliza with para acceder a un objeto
function calcular() {
  //-- Objeto con una propiedad que coincide con la variable global
  const datos = {
    total: 50,
    impuesto: 0.21
  };

  //-- Uso de with que modifica dinámicamente el scope chain
  with (datos) {
    //-- Ambigüedad: ¿se usa el total global o el del objeto?
    return total + (total * impuesto);
  }
}
//-- Ejecución del cálculo
const resultado = calcular();

//-- Salida por consola
console.log('Resultado:', resultado);
```

Codificación recomendada

Acceso explícito a las propiedades del objeto sin modificar el scope chain. Se pretende mostrar cómo una referencia directa y clara elimina ambigüedades, mejora la legibilidad y evita comportamientos no deterministas en la resolución de variables.

```
'use strict';

//-- Definición de una variable global
let total = 100;

//-- Función que accede explícitamente a las propiedades del objeto
function calcular() {
  //-- Objeto con una propiedad que coincide con la variable global
  const datos = {
    total: 50,
    impuesto: 0.21
  };

  //-- Acceso explícito a las propiedades del objeto
  return datos.total + (datos.total * datos.impuesto);
}
//-- Ejecución del cálculo
const resultado = calcular();

//-- Salida por consola
```

```
console.log('Resultado:', resultado);
```

2.1.2.1 Descripción

La declaración `with` modifica dinámicamente la cadena de ámbitos durante la ejecución, alterando el mecanismo normal de resolución de variables. Esta modificación introduce comportamiento no determinista que compromete la predictibilidad del código y genera vulnerabilidades de seguridad.

El uso de `with` crea un **ámbito temporal** que antepone las propiedades del objeto especificado a la cadena de resolución de variables. Esta alteración impide determinar estáticamente qué variables se están referenciando, ya que la resolución depende del contenido dinámico del objeto en tiempo de ejecución.

La ambigüedad en la resolución de nombres puede provocar **accesos no intencionados** a propiedades del objeto o variables del ámbito exterior. Los atacantes pueden explotar esta incertidumbre mediante la manipulación de propiedades del objeto para redirigir referencias de variables hacia valores controlados externamente.

Los motores de optimización no pueden aplicar técnicas de mejora de rendimiento en código que utiliza `with`, ya que la resolución de variables debe realizarse dinámicamente. Esta limitación afecta negativamente al rendimiento y dificulta el análisis estático del código.

La mitigación consiste en **evitar completamente** el uso de `with` y emplear alternativas que mantengan la predictibilidad del ámbito. La asignación directa de objetos a variables locales y el acceso explícito a propiedades proporcionan el mismo resultado funcional sin comprometer la seguridad ni el rendimiento.

El modo estricto prohíbe automáticamente el uso de `with`, proporcionando una capa adicional de protección contra su uso accidental. Esta restricción forma parte de las mejoras de seguridad implementadas en las versiones modernas del lenguaje.

Riesgos asociados

- Introducción de vulnerabilidades de seguridad por confusión en la resolución de variables.
- Dificultad para detectar y prevenir ataques de tipo prototype pollution.
- Creación de código difícil de mantener y auditar debido al comportamiento no determinista del scope chain.
- Mayor probabilidad de errores lógicos derivados de la sombra de variables (variable shadowing).
- Complicaciones en herramientas automáticas de análisis estático y linters, que no pueden determinar fácilmente el flujo de datos.

- Reducción de la posibilidad de optimización por parte de los motores de JavaScript, afectando el rendimiento y la seguridad.
- Incremento del riesgo de uso inadvertido de variables globales.

Buenas prácticas

- Evitar el uso de `with` y acceder explícitamente a las propiedades del objeto para mantener un alcance predecible.

```
//-- Modificación dinámica del scope chain con with
const user = { name: 'Ana', role: 'admin' };
with (user) {
  console.log(name);
}
```

```
//-- Acceso explícito a propiedades sin alterar el scope
const user = { name: 'Ana', role: 'admin' };
console.log(user.name);
```

- Utilizar destructuring para extraer propiedades necesarias en lugar de emplear `with`.

```
//-- Uso de with que genera ambigüedad en variables
const config = { timeout: 5000 };
let timeout = 1000;
with (config) {
  console.log(timeout);
}
```

```
//-- Desestructuración explícita evitando ambigüedad
const config = { timeout: 5000 };
let timeout = 1000;
const { timeout: configTimeout } = config;
console.log(configTimeout);
```

- Restringir el alcance mediante bloques o funciones en lugar de alterar el contexto con `with`.

```
//-- Alteración del contexto que dificulta el análisis estático
const settings = { debug: true };
with (settings) {
  if (debug) {
    console.log('Debug activo');
  }
}
```

```
//-- Uso de bloque con referencia explícita y alcance claro
const settings = { debug: true };
{
  if (settings.debug) {
    console.log('Debug activo');
  }
}
```

2.1.2.2 Estándares metodológicos

- **JS-WASCP-14.2**: Interpreted Code Integrity.
- **Google-JS-5.11**: Disallowed features.

2.1.3 [js.flow.3] - No depender del hoisting implícito de variables y funciones

ID	js.flow.3	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El hoisting permite acceder a variables y funciones antes de su declaración en el código, pero genera **comportamiento impredecible** y errores difíciles de detectar. Las variables declaradas con `var` se inicializan como `undefined` y las funciones se elevan completamente, creando dependencias implícitas que comprometen la **legibilidad y mantenibilidad** del código.

Codificación insegura

Uso de variables y funciones antes de su declaración formal, apoyándose en el hoisting implícito de `var` y de las declaraciones de función. Se busca mostrar cómo este comportamiento puede generar resultados inesperados y dificultar el mantenimiento al depender de un orden de ejecución poco claro.

```
console.log("Inicio del programa");

//-- Se utiliza una variable antes de declararla
console.log("Valor de contador antes de declarar:", contador);

//-- Se invoca una función antes de declararla
saludar();

//-- Declaración de variable con var (afectada por hoisting)
var contador = 10;

//-- Declaración de función tradicional (también afectada por hoisting)
function saludar() {
  //-- Mensaje simple de salida
```

```
  console.log("Hola desde la función");}
  //-- Uso posterior de la variable
console.log("Valor de contador después de declarar:", contador);
```

Codificación recomendada

Declaración explícita de variables con `let` y definición de funciones antes de su uso, evitando depender del hoisting. Se pretende reforzar un orden claro y predecible de ejecución, reduciendo errores y comportamientos no deterministas.

```
console.log("Inicio del programa");

//-- Declaración explícita de variable antes de usarla
let contador = 10;

//-- Declaración de función como constante antes de invocarla
const saludar = function() {
  //-- Mensaje simple de salida
  console.log("Hola desde la función");
};

//-- Uso de la variable después de declararla
console.log("Valor de contador:", contador);

//-- Invocación de la función después de su definición
saludar();
```

2.1.3.1 Descripción

El hoisting es un mecanismo interno que mueve automáticamente las declaraciones de variables y funciones al inicio de su ámbito durante la fase de compilación. Este comportamiento puede generar inconsistencias entre el orden visual del código y su ejecución real, creando condiciones impredecibles que comprometen la seguridad y mantenibilidad del software.

Las declaraciones con `var` y las funciones declaradas con `function` experimentan hoisting completo, mientras que `let` y `const` presentan un comportamiento diferente al crear una zona temporal muerta. Esta disparidad en el tratamiento de declaraciones introduce **comportamiento no determinista** que puede ser explotado o generar errores difíciles de detectar.

La dependencia del hoisting implícito representa un riesgo de seguridad porque permite que el código funcione de manera aparentemente correcta mientras oculta problemas de inicialización, referencias prematuras y conflictos de nombres. Estos problemas pueden manifestarse como vulnerabilidades en tiempo de ejecución o crear vectores de ataque inesperados.

La mitigación efectiva requiere estructurar el código de manera que las declaraciones precedan siempre a su uso, eliminando la dependencia del comportamiento automático del motor. Esto implica adoptar patrones de declaración explícitos y utilizar herramientas de análisis estático que detecten referencias antes de la declaración visual.

El control explícito del orden de declaraciones mejora la **predictibilidad del código** y reduce la superficie de ataque al eliminar comportamientos implícitos que pueden ser malinterpretados tanto por desarrolladores como por sistemas automatizados de análisis de seguridad.

Riesgos asociados

- Comportamiento inesperado del código debido a inicialización de variables en un orden distinto al previsto.
- Acceso a variables antes de su declaración, resultando en valores `undefined` o errores de ejecución.
- Errores difíciles de detectar y depurar por la falta de claridad sobre el momento en que las variables o funciones existen en el contexto de ejecución.
- Explotación de fallas de seguridad por parte de atacantes que aprovechan la confusión creada por el hoisting.
- Inyección de código malicioso en bloques de código que se ejecutan antes de la declaración explícita.
- Fragmentación del código y problemas de mantenibilidad, favoreciendo la inserción de bugs y vulnerabilidades.
- Pérdida de integridad o confidencialidad de datos manipulados por variables cuyo valor aún no ha sido correctamente inicializado.
- Incompatibilidades o comportamientos inconsistentes entre diferentes entornos o versiones de JavaScript.

Buenas prácticas

- Declarar siempre las variables al inicio de su ámbito utilizando `let` o `const` antes de usarlas.

```
//-- Uso de variable antes de su declaración (dependencia de hoisting)
console.log(total);
var total = 10;
```

```
//-- Declaración explícita antes de su uso con alcance de bloque
let totalSeguro = 10;
```

```
console.log(totalSeguro);
```

- Evitar el uso de `var` y preferir `let` o `const` para impedir comportamientos inesperados por elevación de declaraciones.

```
//-- Variable declarada con var dentro de bloque accesible fuera de él
if (true) {
  var estado = "activo";
}
console.log(estado);
```

```
//-- Variable declarada con let limitada al bloque correspondiente
if (true) {
  let estadoSeguro = "activo";
  console.log(estadoSeguro);
}
```

- Definir funciones antes de invocarlas y evitar depender de la elevación implícita de declaraciones.

```
//-- Invocación de función antes de su definición (hoisting implícito)
procesar();
function procesar() {
  console.log("Procesando");
}
```

```
//-- Definición previa de función asignada a constante antes de su uso
const procesarSeguro = function() {
  console.log("Procesando");
};
procesarSeguro();
```

- Inicializar explícitamente las variables en el momento de su declaración para evitar valores `undefined` inesperados.

```
//-- Declaración sin inicialización y uso previo
console.log(config);
var config;
```

```
//-- Declaración con inicialización explícita antes de su uso
let configSeguro = {};
console.log(configSeguro);
```

2.1.3.2 Estándares metodológicos

- **Airbnb-JS-14**: Hoisting.
- **Google-JS-5.1**: Local variable declarations.

2.1.4 [js.flow.4] - Usar siempre bloques explícitos en estructuras de control

ID	js.flow.4	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las estructuras de control sin bloques explícitos pueden generar **comportamientos inesperados** cuando se modifican o mantienen. La ausencia de llaves delimita incorrectamente el alcance de las instrucciones, provocando que sentencias adicionales se ejecuten fuera del contexto previsto o que modificaciones posteriores alteren la lógica original sin generar errores sintácticos visibles.

Codificación insegura

Control de flujo con una estructura condicional donde se omiten llaves en un bloque `if`. Se busca mostrar cómo una instrucción adicional puede ejecutarse siempre, aunque visualmente parezca estar dentro de la condición, generando comportamientos inesperados y posibles fallos lógicos.

```
const isAdmin = false;

//-- Simula una acción sensible que solo debería ejecutar un administrador
function deleteAllUsers() {
  console.log("Usuarios eliminados");
}
//-- Estructura condicional sin llaves explícitas
if (isAdmin)
  deleteAllUsers();
  //-- Esta línea se ejecuta siempre, aunque no sea administrador al no haber
llaves de bloque
  console.log("Acción registrada en el sistema");

console.log("Proceso finalizado");
```

Codificación recomendada

Control de flujo con la misma lógica, pero utilizando bloques explícitos con llaves en la estructura `if`. Se busca garantizar que todas las instrucciones relacionadas con la condición estén claramente delimitadas y evitar ejecuciones no deseadas.

```
const isAdmin = false;
```

```
//-- Simula una acción sensible que solo debería ejecutar un administrador
function deleteAllUsers() {
  console.log("Usuarios eliminados");
}
//-- Estructura condicional con bloque explícito
if (isAdmin) {
  //-- Solo se ejecuta si el usuario es administrador
  deleteAllUsers();
  console.log("Acción registrada en el sistema");
}
//-- Flujo del programa claramente definido
console.log("Proceso finalizado");
```

2.1.4.1 Descripción

Las estructuras de control requieren el uso de bloques delimitados por llaves para definir claramente el alcance de las instrucciones que deben ejecutarse. La omisión de estos bloques puede provocar que el comportamiento del programa difiera de las intenciones del desarrollador, especialmente cuando se realizan modificaciones posteriores al código.

El problema surge cuando las estructuras como `if`, `else`, `for`, `while` o `do-while` se escriben sin llaves, permitiendo que solo la primera instrucción siguiente forme parte del bloque de control. Esta sintaxis puede generar confusión visual y errores lógicos cuando se añaden nuevas líneas de código.

Los riesgos asociados incluyen la ejecución no intencionada de código fuera del contexto esperado, lo que puede comprometer la lógica de negocio y crear vulnerabilidades de seguridad. La indentación visual del código puede crear una falsa sensación de agrupación que no corresponde con la estructura real del programa.

La mitigación de estos problemas se logra mediante el uso consistente de llaves en todas las estructuras de control, independientemente del número de instrucciones que contengan. Esta práctica mejora la legibilidad del código y reduce significativamente la probabilidad de introducir errores durante el mantenimiento. Las herramientas de análisis estático y los formateadores de código pueden automatizar la detección y corrección de estas situaciones.

La aplicación sistemática de esta regla contribuye a mantener un código más **predecible** y **mantenible**, reduciendo los errores relacionados con la estructura de control del programa.

Riesgos asociados

- Ejecución de código no intencionado debido a ambigüedad en el alcance de la instrucción dentro de la estructura de control.

- Introducción de errores lógicos difíciles de detectar, especialmente al modificar o extender el código posteriormente.
- Vulnerabilidad a ataques de manipulación del flujo de ejecución, como elevación de privilegios o bypass de validaciones de seguridad.
- Inyección accidental de código malicioso si un atacante logra influenciar la estructura condicional en ausencia de bloques explícitos.
- Posibilidad de que funciones sensibles se ejecuten fuera del contexto seguro previsto.
- Reducción de la legibilidad y mantenibilidad del código, lo que dificulta la auditoría y detección de fallos de seguridad.

Buenas prácticas

- Usar siempre llaves en estructuras `if`, incluso cuando solo haya una instrucción.

```
//-- if sin llaves puede generar ejecución no intencionada
if (isAdmin)
  grantAccess();
  logAccess();
```

```
//-- if con bloque explícito delimitando claramente las instrucciones
if (isAdmin) {
  grantAccess();
  logAccess();
}
```

- Encerrar siempre las estructuras `else` en bloques para evitar ambigüedades.

```
//-- else sin llaves puede asociarse a un if diferente
if (isValid)
  if (isAdmin)
    grantAccess();
else
  denyAccess();
```

```
//-- else claramente asociado mediante bloques explícitos
if (isValid) {
  if (isAdmin) {
    grantAccess();
  }
} else {
  denyAccess();
}
```

- Utilizar bloques en bucles `for` y `while` aunque contengan una sola instrucción.

```
//-- bucle sin llaves expone a errores al añadir nuevas líneas
for (let i = 0; i < users.length; i++)
  processUser(users[i]);
  auditUser(users[i]);
```

```
//-- bucle con bloque que delimita las acciones iterativas
for (let i = 0; i < users.length; i++) {
  processUser(users[i]);
  auditUser(users[i]);
}
```

- Definir bloques explícitos en estructuras `while` para evitar ejecuciones fuera del ciclo.

```
//-- while sin llaves puede ejecutar código fuera del ciclo
while (hasAccess)
  refreshSession();
  logSession();
```

```
//-- while con bloque que agrupa correctamente las instrucciones
while (hasAccess) {
  refreshSession();
  logSession();
}
```

2.1.4.2 Estándares metodológicos

- **Airbnb-JS-16**: Blocks.
- **Google-JS-4.1**: Braces.

2.2 Gestión de memoria

2.2.1 [js.mem.1] - Evitar el uso implícito de variables globales

ID	**js.mem.1**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

La omisión de declaraciones explícitas de variables puede generar **variables globales implícitas** que contaminan el espacio de nombres global. Este comportamiento introduce **efectos secundarios no deseados** y dificulta el mantenimiento del código,

especialmente en aplicaciones complejas donde múltiples módulos pueden interferir entre sí.

Codificación insegura

Demostración de cómo la omisión de palabras clave al declarar variables dentro de una función provoca la creación implícita de variables globales, generando efectos secundarios y corrupción del estado compartido. Se busca observar cómo una variable puede ser modificada desde cualquier parte del código sin control.

```
//-- Simulación de un módulo que calcula el total de una compra
function calcularTotal(precio, cantidad) {
  //-- Asignación sin let, const o var (crea variable global implícita)
  total = precio * cantidad;
  return total;
}
//-- Otra función que modifica la misma variable global sin intención
function aplicarDescuento(porcentaje) {
  //-- Se reutiliza la variable global sin declararla
  total = total - (total * porcentaje);
  return total;
}
//-- Ejecución del flujo principal
console.log("Total inicial:", calcularTotal(100, 2));
console.log("Total con descuento:", aplicarDescuento(0.1));

//-- Acceso directo a la variable global desde el ámbito global
console.log("Valor final de total en ámbito global:", total);
```

Codificación recomendada

Uso explícito de declaraciones con `let` y activación del modo estricto para evitar la creación de variables globales accidentales. Se pretende mostrar cómo limitar el alcance de las variables y proteger el estado interno de cada función.

```
//-- Activación del modo estricto para evitar variables globales implícitas
"use strict";

//-- Simulación de un módulo que calcula el total de una compra
function calcularTotal(precio, cantidad) {
  //-- Declaración explícita de variable local
  let total = precio * cantidad;
  return total;
}
//-- Función que recibe el total como parámetro y no depende de estado global
function aplicarDescuento(total, porcentaje) {
  //-- Uso de variable local para el nuevo cálculo
  let totalConDescuento = total - (total * porcentaje);
```

```
  return totalConDescuento;
}
//-- Ejecución del flujo principal sin exponer variables al ámbito global
let totalInicial = calcularTotal(100, 2);
console.log("Total inicial:", totalInicial);

let totalFinal = aplicarDescuento(totalInicial, 0.1);
console.log("Total con descuento:", totalFinal);
```

2.2.1.1 Descripción

La declaración explícita de variables mediante las palabras clave `let`, `const` o `var` constituye una práctica obligatoria para mantener el control sobre el ámbito de las variables. Cuando se omite esta declaración, el motor de ejecución asigna automáticamente la variable al objeto global, creando dependencias no intencionadas y modificaciones imprevistas del estado compartido.

El **ámbito implícito** genera vulnerabilidades de seguridad al permitir que variables locales se conviertan inadvertidamente en propiedades del objeto global. Esta situación facilita el acceso no autorizado a datos sensibles y la manipulación del estado de la aplicación desde contextos externos o módulos no relacionados.

Los riesgos asociados incluyen la **corrupción del estado global**, conflictos de nombres entre variables de diferentes módulos, y la exposición accidental de información confidencial. Estas condiciones pueden derivar en comportamientos impredecibles de la aplicación y brechas de seguridad explotables.

La mitigación efectiva requiere el establecimiento de políticas de declaración explícita en todas las asignaciones de variables. El uso de herramientas de análisis estático y la configuración de entornos en **modo estricto** proporcionan mecanismos adicionales de detección y prevención de estas situaciones problemáticas.

La implementación de esta regla reduce significativamente la superficie de ataque relacionada con la manipulación no autorizada del estado global y mejora la predictibilidad del comportamiento del código en entornos de ejecución diversos.

Riesgos asociados

- Colisiones y sobrescritura accidental de variables globales, afectando el comportamiento de otras partes del programa.
- Exposición involuntaria de datos sensibles al scope global, facilitando accesos no autorizados.
- Dificultad para depurar errores debido a cambios inesperados en el estado global.
- Incremento de vulnerabilidades como Cross-Site Scripting (XSS) al exponer datos

críticos en el ámbito global.

- Problemas de compatibilidad y mantenimiento, especialmente en aplicaciones grandes o desarrollos colaborativos.
- Imposibilidad de detectar fácilmente dependencias y efectos colaterales, complicando la lectura y comprensión del código.
- Facilita que bibliotecas de terceros o scripts maliciosos manipulen el estado global a conveniencia.

Buenas prácticas

- Declarar siempre las variables con `let` o `const` para evitar la creación implícita de variables globales.

```
//-- Asignación sin declaración previa genera variable global implícita
function calcular() {
  resultado = 10 * 2;
}
```

```
//-- Declaración explícita limita el alcance al contexto definido
function calcular() {
  let resultado = 10 * 2;
}
```

- Usar `const` por defecto y reservar `let` solo cuando sea necesaria la reasignación.

```
//-- Variable global implícita reutilizada accidentalmente
contador = 0;
function incrementar() {
  contador++;
}
```

```
//-- Constante declarada explícitamente dentro del alcance requerido
function incrementar() {
  let contador = 0;
  contador++;
}
```

- Activar el modo estricto (`"use strict"`) para impedir la creación accidental de variables globales.

```
//-- Sin modo estricto permite variable global implícita
function guardar() {
  dato = "secreto";
}
```

```
//-- Modo estricto fuerza declaración explícita de variables
"use strict";
function guardar() {
  let dato = "secreto";
}
```

- Encapsular el código en funciones o módulos para evitar la contaminación del espacio global.

```
//-- Variable definida en el ámbito global
let token = "ABC123";
```

```
//-- Variable encapsulada dentro de una función autoejecutable
(function() {
  let token = "ABC123";
})();
```

2.2.1.2 Estándares metodológicos

- **Airbnb-JS-13**: Variables.
- **Google-JS-5.1**: Local variable declarations.

2.2.2 [js.mem.2] - No reutilizar referencias a objetos mutables compartidos entre contextos asíncronos sin un control explícito

ID	**js.mem.2**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los objetos mutables compartidos entre diferentes contextos asíncronos pueden generar **condiciones de carrera** y estados inconsistentes cuando múltiples operaciones asíncronas modifican simultáneamente las mismas referencias. Esta situación compromete la integridad de los datos y produce comportamientos impredecibles en aplicaciones que manejan concurrencia.

Codificación insegura

Gestión de un objeto compartido entre varias operaciones asíncronas que modifican su estado. Se pretende mostrar cómo el uso de una misma referencia mutable en distintos contextos puede generar resultados inesperados debido a condiciones de carrera y falta de control sobre el estado compartido.

```
const sharedData = { counter: 0 };
```

```
//-- Simula una operación asíncrona que modifica el objeto compartido
function incrementAsync() {
  setTimeout(() => {
    //-- Se modifica directamente el objeto compartido
    sharedData.counter++;
    console.log("Increment:", sharedData.counter);
  }, Math.random() * 100);
}
//-- Se lanzan múltiples operaciones asíncronas sobre la misma referencia
for (let i = 0; i < 5; i++) {
  incrementAsync();
}
//-- Se imprime el valor final tras un tiempo de espera
setTimeout(() => {
  console.log("Final:", sharedData.counter);
}, 500);
```

Codificación recomendada

Uso controlado de datos en operaciones asíncronas evitando compartir directamente la misma referencia mutable. Se busca garantizar consistencia creando copias independientes del estado para cada contexto asíncrono y consolidando los resultados de forma controlada.

```
const sharedData = { counter: 0 };

//-- Simula una operación asíncrona que trabaja sobre una copia del estado
function incrementAsyncSafe(baseState) {
  //-- Se crea una copia explícita para evitar modificar la referencia original
  const localCopy = { ...baseState };

  return new Promise((resolve) => {
    setTimeout(() => {
      //-- Se modifica únicamente la copia local
      localCopy.counter++;
      resolve(localCopy.counter);
    }, Math.random() * 100);
  });
}
//-- Se ejecutan múltiples operaciones controladas
async function run() {
  const promises = [];

  for (let i = 0; i < 5; i++) {
    promises.push(incrementAsyncSafe(sharedData));
  }
  //-- Consolidación explícita de resultados
  const results = await Promise.all(promises);
```

```
  sharedData.counter += results.length;

  console.log("Final:", sharedData.counter);
}
run();
```

2.2.2.1 Descripción

El problema se manifiesta especialmente en aplicaciones que manejan múltiples `Promise`, `callbacks`, o funciones `async/await` que operan sobre estructuras de datos compartidas como arrays, objetos o mapas. La naturaleza no determinista de la ejecución asíncrona hace que el orden de las modificaciones sea impredecible.

Los riesgos incluyen la corrupción de datos, comportamientos inconsistentes de la aplicación y errores difíciles de reproducir y depurar. Estos problemas se agravan en entornos de alta concurrencia donde múltiples operaciones asíncronas se ejecutan simultáneamente.

La mitigación requiere implementar estrategias de control explícito sobre el acceso a objetos compartidos. Esto incluye el uso de técnicas de inmutabilidad, donde se crean nuevas instancias en lugar de modificar objetos existentes, y la implementación de mecanismos de sincronización que serialicen el acceso a recursos compartidos.

También resulta efectivo diseñar arquitecturas que minimicen el uso compartido de estado mutable, favoreciendo patrones que aíslen el estado dentro de contextos específicos o que utilicen canales de comunicación explícitos entre componentes asíncronos.

Riesgos asociados

- Condiciones de carrera por acceso concurrente no controlado a los mismos datos.
- Corrupción de datos al modificarse el objeto por diferentes contextos asíncronos simultáneamente.
- Dificultad para depurar errores debido a estados difíciles de reproducir y no deterministas.
- Filtrado accidental de información sensible si el objeto es expuesto en contextos no seguros.
- Introducción de vulnerabilidades como acceso no autorizado, denegación de servicio o escalado de privilegios.
- Pérdida de integridad en los datos procesados durante la ejecución asíncrona.
- Resultados inesperados o inconsistentes en la lógica de negocio por manipulación no controlada del objeto.

- Exposición inadvertida de información entre usuarios en aplicaciones multiusuario.

Buenas prácticas

- Evitar compartir el mismo objeto mutable entre múltiples operaciones asíncronas; crear copias independientes para cada flujo.

```
//-- Uso de un mismo objeto mutable en operaciones asíncronas
const config = { retries: 3 };
setTimeout(() => {
  config.retries--;
}, 100);
setTimeout(() => {
  console.log(config.retries);
}, 200);
```

```
//-- Creación de copias independientes para cada operación
const baseConfig = { retries: 3 };
setTimeout(() => {
  const localConfig = { ...baseConfig };
  localConfig.retries--;
}, 100);
setTimeout(() => {
  const localConfig = { ...baseConfig };
  console.log(localConfig.retries);
}, 200);
```

- Aplicar inmutabilidad a objetos compartidos mediante congelación o estructuras inmutables.

```
//-- Modificación directa de objeto compartido
const session = { user: "admin" };
async function updateSession() {
  session.user = "guest";
}
```

```
//-- Uso de Object.freeze para evitar modificaciones
const session = Object.freeze({ user: "admin" });
async function updateSession() {
  const newSession = { ...session, user: "guest" };
}
```

- Encapsular el acceso a objetos compartidos usando mecanismos de sincronización lógica (por ejemplo, colas o locks).

```
//-- Acceso concurrente sin control a recurso compartido
```

```
let counter = { value: 0 };
async function increment() {
  counter.value++;
}
```

```
//-- Serialización del acceso mediante promesa encadenada
let counter = { value: 0 };
let lock = Promise.resolve();
function increment() {
  lock = lock.then(() => {
    counter.value++;
  });
  return lock;
}
```

- Evitar exponer referencias internas de objetos mutables; devolver copias defensivas.

```
//-- Exposición directa de referencia interna mutable
const store = { data: [] };
function getData() {
  return store.data;
}
```

```
//-- Retorno de copia defensiva para evitar mutaciones externas
const store = { data: [] };
function getData() {
  return [...store.data];
}
```

2.2.2.2 Estándares metodológicos

- **JS-WASCP-14.4**: Concurrency.
- **Airbnb-JS-2**: References.

2.2.3 [js.mem.3] - Liberar referencias a objetos que ya no se usan

ID	**js.mem.3**	**Uso**	**Recomendado**
CWSS Score	**74**	**Impacto técnico**	**Alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las **fugas de memoria** ocurren cuando el código mantiene referencias innecesarias a objetos que ya no requiere para su funcionamiento. Estas referencias impiden que el **recolector de basura** libere la memoria ocupada, causando un consumo progresivo de

recursos que puede degradar el rendimiento de la aplicación o provocar fallos del sistema.

Codificación insegura

Gestión de objetos en memoria dentro de un proceso Node.js que crea estructuras grandes en un bucle. Se busca observar cómo mantener referencias innecesarias impide que el recolector de basura libere memoria.

```
const cache = [];

function procesarDatos() {
  //-- Simula la creación de un objeto grande en memoria
  const datos = new Array(1000000).fill("dato");

  //-- Se almacena en un array global sin necesidad real
  cache.push(datos);
}
function ejecutar() {
  //-- Ejecuta múltiples veces la creación de datos
  for (let i = 0; i < 50; i++) {
    procesarDatos();
  }
  //-- Se muestra el tamaño del cache que sigue creciendo
  console.log("Elementos en cache:", cache.length);
}
ejecutar();
```

Codificación recomendada

Gestión de objetos en memoria dentro de un proceso Node.js que crea estructuras grandes en un bucle. Se busca observar cómo eliminar referencias innecesarias permite que el recolector de basura libere memoria y evita el crecimiento innecesario del consumo.

```
function procesarDatos() {
  //-- Simula la creación de un objeto grande en memoria
  let datos = new Array(1000000).fill("dato");

  //-- Se utiliza el objeto para alguna operación puntual
  console.log("Procesando lote de datos");

  //-- Se elimina explícitamente la referencia cuando ya no se necesita
  datos = null;
}
function ejecutar() {
  //-- Ejecuta múltiples veces la creación controlada de datos
  for (let i = 0; i < 50; i++) {
```

```
    procesarDatos();
  }
  //-- No se mantienen referencias globales innecesarias
  console.log("Ejecución finalizada");
}
ejecutar();
```

2.2.3.1 Descripción

Las referencias circulares, los **event listeners** no removidos, las variables globales persistentes y los closures que capturan objetos de gran tamaño representan las principales fuentes de retención no deseada de memoria. Estos patrones impiden que el recolector de basura identifique correctamente los objetos elegibles para liberación.

El problema se agrava en aplicaciones de larga duración, como **Single Page Applications**, donde la acumulación progresiva de referencias no liberadas puede consumir cantidades significativas de memoria del sistema. Los componentes dinámicos que se crean y destruyen frecuentemente son especialmente susceptibles a este tipo de problemas.

La mitigación efectiva requiere establecer referencias a `null` cuando los objetos ya no son necesarios, remover explícitamente los event listeners antes de eliminar elementos del DOM, y evitar la creación de referencias circulares entre objetos. El uso de **WeakMap** y **WeakSet** proporciona alternativas que permiten referencias débiles que no interfieren con la recolección de basura.

Las herramientas de desarrollo del navegador permiten monitorear el uso de memoria y detectar posibles fugas mediante análisis de heap y perfiles de memoria. La implementación de patrones de limpieza sistemática y la revisión regular del consumo de memoria constituyen prácticas necesarias para mantener la estabilidad de la aplicación.

Riesgos asociados

- Fugas de memoria persistentes que deterioran gradualmente el rendimiento de la aplicación.
- Consumo innecesario de recursos, pudiendo llevar a la falta de disponibilidad de memoria para otras operaciones.
- Posible degradación progresiva en la velocidad de respuesta del sistema, especialmente en aplicaciones de larga duración o alta carga.
- Eventuales bloqueos o caídas del servidor/aplicación debido a agotamiento de recursos.
- Exposición a ataques de denegación de servicio (DoS) facilitada por la saturación de recursos causada por fugas de memoria.

- Dificultad en la detección y corrección de errores, incrementando el coste de mantenimiento.
- Reducción de la escalabilidad, ya que la aplicación utiliza más memoria de la necesaria, limitando la capacidad para manejar múltiples usuarios o procesos concurrentes.
- En entornos compartidos, posible impacto negativo en otros servicios o aplicaciones ejecutándose en el mismo entorno debido al agotamiento de memoria compartida.

Buenas prácticas

- Eliminar referencias a objetos grandes cuando ya no se necesiten para facilitar la recolección de basura.

```
//-- Se mantiene referencia innecesaria a un objeto grande
let datos = new Array(1000000).fill("valor");
procesar(datos);
```

```
//-- Se libera la referencia tras su uso
let datos2 = new Array(1000000).fill("valor");
procesar(datos2);
datos2 = null;
```

- Evitar variables globales que mantengan objetos en memoria más tiempo del necesario.

```
//-- Variable global mantiene referencia permanente
let cacheGlobal = {};
function guardar(id, valor) {
  cacheGlobal[id] = valor;
}
```

```
//-- Variable limitada al ámbito necesario
function guardarSeguro(id, valor) {
  let cacheLocal = {};
  cacheLocal[id] = valor;
}
```

- Eliminar manejadores de eventos cuando ya no sean necesarios para evitar retención de objetos.

```
//-- Evento añadido sin eliminación posterior
let boton = document.getElementById("miBoton");
boton.addEventListener("click", function() {
  console.log("Click");
```

```
});
```

```
//-- Evento eliminado cuando deja de ser necesario
let boton2 = document.getElementById("miBoton");
function manejarClick() {
  console.log("Click");
}
boton2.addEventListener("click", manejarClick);
boton2.removeEventListener("click", manejarClick);
```

- Vaciar estructuras de datos reutilizables cuando ya no contengan información útil.

```
//-- Array conserva elementos innecesarios
let lista = ["a", "b", "c"];
procesar(lista);
```

```
//-- Array limpiado tras su uso
let lista2 = ["a", "b", "c"];
procesar(lista2);
lista2.length = 0;
```

2.2.3.2 Estándares metodológicos

- **JS-WASCP-9**: Data Protection.
- **MDN Web Docs JS-18**: Resource management.

2.3 Validaciones

2.3.1 [js.val.1] - No construir código ejecutable a partir de entrada externa sin validación estricta

ID	js.val.1	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La construcción de código ejecutable a partir de datos externos representa una vulnerabilidad crítica que permite la **inyección de código malicioso**. Funciones como `eval()`, `Function()` y `setTimeout()` con strings pueden ejecutar contenido no confiable, comprometiendo la seguridad de la aplicación y exponiendo el sistema a ataques de ejecución remota de código.

Codificación insegura

Se muestra cómo una aplicación de línea de comandos ejecuta una operación indicada por el usuario. El objetivo es entender el riesgo de construir y ejecutar código dinámicamente a partir de datos externos sin aplicar validaciones estrictas, lo que puede derivar en ejecución de código arbitrario.

```
const userInput = process.argv[2];
//-- Se obtiene la entrada del usuario desde la línea de comandos
//-- Se construye dinámicamente una expresión usando la entrada externa
const dynamicCode = "console.log('Resultado:', " + userInput + ")";

//-- Se ejecuta el código dinámico sin ningún tipo de validación
eval(dynamicCode);

//-- Ejemplo de uso inseguro: node app.js "2 + 2" o algo malicioso como
"process.exit()"
```

Codificación recomendada

Se presenta la misma funcionalidad de cálculo por línea de comandos, pero aplicando validación estricta y evitando la construcción de código dinámico. Se aprende a restringir las entradas a valores permitidos y a utilizar estructuras seguras en lugar de evaluar texto como código.

```
const userInput = process.argv[2];
//-- Se obtiene la entrada del usuario desde la línea de comandos
//-- Se define una lista blanca de operaciones permitidas
const allowedOperations = {
  "sumar": (a, b) => a + b,
  "restar": (a, b) => a - b
};

//-- Se espera un formato controlado: operacion,numero1,numero2
if (!userInput) {
  console.error("Debe indicar una operación en formato:
operacion,numero1,numero2");
  process.exit(1);
}
const parts = userInput.split(",");

//-- Se valida que la entrada tenga exactamente tres partes
if (parts.length !== 3) {
  console.error("Formato inválido");
  process.exit(1);
}
const [operation, aStr, bStr] = parts;
const a = Number(aStr);
const b = Number(bStr);
```

```
//-- Se valida que la operación esté en la lista blanca y que los números sean
válidos
if (!allowedOperations[operation] || Number.isNaN(a) || Number.isNaN(b)) {
  console.error("Operación o parámetros no permitidos");
  process.exit(1);
}
//-- Se ejecuta únicamente la función previamente definida y validada
const result = allowedOperations[operation](a, b);
console.log("Resultado:", result);
```

2.3.1.1 Descripción

La construcción de código ejecutable a partir de entrada externa representa una de las vulnerabilidades más críticas en el desarrollo de aplicaciones. Esta práctica permite que datos no confiables se transformen en instrucciones que el intérprete ejecuta directamente, creando un vector de ataque directo hacia el sistema.

El problema surge cuando se utilizan funciones como `eval()`, `Function()`, `setTimeout()` con cadenas de texto, o `setInterval()` para procesar datos que provienen de fuentes externas como formularios web, parámetros de URL, cookies, o respuestas de servicios web. Estas funciones interpretan las cadenas de texto como código válido y las ejecutan en el contexto actual de la aplicación.

Los atacantes pueden explotar esta funcionalidad inyectando código malicioso dentro de los datos de entrada. Una vez que este código se ejecuta, puede acceder a variables globales, manipular el DOM, realizar peticiones no autorizadas, o extraer información sensible del navegador del usuario.

La **validación estricta** implica verificar que los datos de entrada cumplan exactamente con el formato, tipo y rango esperados antes de cualquier procesamiento. Sin embargo, incluso con validación, la construcción dinámica de código mantiene riesgos inherentes debido a la complejidad de predecir todas las posibles combinaciones maliciosas.

Las estrategias de mitigación se centran en evitar completamente la generación dinámica de código cuando sea posible. Se pueden utilizar estructuras de datos como objetos o arrays para mapear entradas a funciones predefinidas, eliminando la necesidad de interpretación dinámica. Cuando la generación de código resulte inevitable, se debe implementar una validación exhaustiva que incluya listas blancas de caracteres permitidos, sanitización de entrada y ejecución en contextos con privilegios limitados.

La aplicación de esta regla requiere revisar todas las instancias donde se construye código a partir de datos externos y evaluar si existen alternativas más seguras para lograr la misma funcionalidad.

Riesgos asociados

- Ejecución de código malicioso (Remote Code Execution) por inyectar entradas manipuladas.
- Vulnerabilidad a ataques de Cross-Site Scripting (XSS).
- Exposición a ataques de Inyección de Código JavaScript.
- Compromiso de integridad y confidencialidad de datos sensibles.
- Desvío de lógica de aplicación mediante entradas manipuladas.
- Pérdida de control sobre el flujo de ejecución del programa.
- Riesgo de denegación de servicio (DoS) por ejecución de código inesperado.
- Exposición de información interna o rutas de archivos sensibles (Information Disclosure).
- Mayor dificultad en la detección y remediación de vulnerabilidades.
- Incumplimiento de normativas o estándares de seguridad.

Buenas prácticas

- Evitar el uso de `eval` con datos provenientes de entrada externa.

```
//-- Uso directo de entrada externa para construir y ejecutar código dinámicamente
const userInput = req.query.expresion;
const resultado = eval(userInput);
```

```
//-- Validación estricta y uso de funciones seguras predefinidas
const operacionesPermitidas = {
  sumar: (a, b) => a + b,
  restar: (a, b) => a - b
};
const operacion = operacionesPermitidas[req.query.operacion];
if (operacion) {
  const resultado = operacion(2, 3);
}
```

- No utilizar el constructor `Function` con cadenas generadas dinámicamente a partir de datos externos.

```
//-- Creación dinámica de función a partir de entrada sin validar
const filtro = new Function("item", "return item." + req.body.campo);
const valor = filtro(objeto);
```

```
//-- Lista blanca de propiedades permitidas antes de acceder dinámicamente
const camposPermitidos = ["nombre", "edad"];
```

```
if (camposPermitidos.includes(req.body.campo)) {
  const valor = objeto[req.body.campo];
}
```

- Evitar ejecutar comandos del sistema construidos con datos externos sin validación.

```
//-- Construcción directa de comando del sistema con entrada externa
const { exec } = require("child_process");
exec("ls " + req.query.directorio);
```

```
//-- Validación estricta y uso de argumentos separados
const { execFile } = require("child_process");
const directoriosPermitidos = ["/var/www", "/tmp"];
if (directoriosPermitidos.includes(req.query.directorio)) {
  execFile("ls", [req.query.directorio]);
}
```

- No insertar directamente contenido externo en métodos que interpretan código o HTML dinámico.

```
//-- Inserción directa de entrada externa que puede contener código ejecutable
element.innerHTML = req.query.contenido;
```

```
//-- Uso de texto plano para evitar interpretación de código
element.textContent = req.query.contenido;
```

2.3.1.2 Estándares metodológicos

- **JS-WASCP-14.2**: Interpreted Code Integrity.
- **JS-WASCP-2.1.1**: Data Sources.
- **Airbnb-JS-7**: Functions.

2.3.2 [js.val.2] - Validar y sanear todas las entradas externas antes de su uso

ID	**js.val.2**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las aplicaciones reciben datos de múltiples fuentes externas como formularios, APIs, parámetros de URL y archivos subidos por usuarios. Estos datos pueden contener contenido malicioso o formato inesperado que comprometa la **seguridad** y **estabilidad** del sistema. Sin validación adecuada, los atacantes pueden ejecutar **inyecciones de**

código, manipular el comportamiento de la aplicación o corromper el estado interno de los datos.

Codificación insegura

Se muestra cómo una entrada proporcionada por el usuario puede influir directamente en el flujo del programa cuando no se valida ni se sanea. El objetivo es observar cómo el uso directo de datos externos en funciones dinámicas puede permitir inyecciones de código y alterar el comportamiento esperado de la aplicación.

```
/-- Programa vulnerable a inyección de código usando eval

const userInput = process.argv[2];

//-- Se verifica que el usuario haya proporcionado un argumento
if (!userInput) {
  console.log("Uso: node app.js <nombre>");
  process.exit(1);
}
//-- Uso inseguro de la entrada del usuario dentro de eval
const script = "console.log('Hola " + userInput + "')";

//-- La ejecución directa permite inyección de código
eval(script);
```

Codificación recomendada

Se presenta el mismo escenario, pero aplicando validación estricta de la entrada antes de utilizarla. Se restringe el formato permitido mediante una expresión regular y se elimina el uso de funciones dinámicas como `eval`, evitando así inyecciones y reduciendo el riesgo de corrupción del estado de la aplicación.

```
//-- Programa seguro validando y saneando la entrada del usuario
const userInput = process.argv[2];

//-- Se verifica que el usuario haya proporcionado un argumento
if (!userInput) {
  console.log("Uso: node app.js <nombre>");
  process.exit(1);
}
//-- Validación estricta: solo letras y espacios permitidos
const validPattern = /^[a-zA-Z\s]+$/;

if (!validPattern.test(userInput)) {
  console.error("Entrada no válida. Solo se permiten letras y espacios.");
  process.exit(1);
}
//-- Uso seguro sin funciones dinámicas
```

```
console.log("Hola " + userInput);
```

2.3.2.1 Descripción

La validación y saneamiento de entradas externas constituye una práctica obligatoria para **mantener la integridad del sistema** y prevenir manipulaciones maliciosas. Toda información que provenga de fuentes externas debe ser verificada antes de su procesamiento o almacenamiento.

Las entradas externas incluyen datos provenientes de formularios web, parámetros de URL, cabeceras HTTP, cookies, APIs externas, archivos subidos por usuarios y cualquier información recibida desde el cliente o sistemas de terceros. Estos datos pueden contener contenido malicioso diseñado para explotar vulnerabilidades en la aplicación.

El proceso de validación verifica que los datos cumplan con los criterios esperados en términos de tipo, formato, longitud y rango de valores permitidos. Esta verificación debe realizarse tanto en el lado cliente como en el servidor, siendo la validación del servidor la que proporciona la protección real.

El saneamiento transforma o filtra los datos de entrada para eliminar o neutralizar elementos potencialmente peligrosos. Este proceso puede incluir la eliminación de caracteres especiales, la codificación de entidades HTML, el escape de caracteres de control de base de datos, o la normalización de formatos de datos.

Los riesgos asociados al manejo inadecuado de entradas externas incluyen **ataques de inyección**, corrupción de datos, **ejecución de código malicioso** y compromiso de la seguridad del sistema. Estos vectores de ataque pueden resultar en acceso no autorizado, pérdida de información sensible o alteración del comportamiento esperado de la aplicación.

La mitigación efectiva requiere implementar controles de validación estrictos, utilizar listas blancas de valores permitidos cuando sea posible, aplicar técnicas de saneamiento apropiadas para cada contexto de uso, y mantener una separación clara entre datos y código ejecutable. La implementación de estas medidas debe ser sistemática y aplicarse de forma consistente en todos los puntos de entrada de la aplicación.

Riesgos asociados

- Inyección de código (por ejemplo, inyección de scripts en HTML o JavaScript – XSS).
- Ejecución de comandos no autorizados en el sistema (Command Injection).
- Acceso y manipulación no autorizada de bases de datos (SQL Injection o NoSQL Injection).
- Corrupción de datos o del estado de la aplicación (por datos inesperados o maliciosos).

- Saltos de control de acceso y escalación de privilegios.
- Fallo de funcionamiento o denegación de servicio (DoS) por entrada inesperada o malformada.
- Exposición de información sensible mediante errores o comportamientos inesperados.
- Robo de sesión o suplantación de identidad (Session Hijacking/Fixation mediante payloads maliciosos).
- Bifurcaciones lógicas o bypass de validaciones críticas de seguridad.
- Problemas de reputación y legales por filtración o manipulación de datos.

Buenas prácticas

- Validar y sanear datos antes de construir consultas dinámicas para evitar inyección de código.

```
//-- Construcción directa de consulta con entrada externa
const username = req.query.user;
const query = "SELECT * FROM users WHERE name = '" + username + "'";
db.execute(query);
```

```
//-- Uso de consultas parametrizadas y validación básica de formato
const username = String(req.query.user).trim();
if (!/^[a-zA-Z0-9_]{1,30}$/.test(username)) { throw new Error('Invalid input'); }
db.execute("SELECT * FROM users WHERE name = ?", [username]);
```

- Escapar o sanear contenido antes de insertarlo en el DOM para prevenir XSS.

```
//-- Inserción directa de entrada externa en el DOM
const comment = location.hash.substring(1);
document.getElementById('output').innerHTML = comment;
```

```
//-- Uso de textContent para evitar interpretación de HTML
const comment = String(location.hash.substring(1));
document.getElementById('output').textContent = comment;
```

- Validar tipos y rangos de datos numéricos antes de operar con ellos para evitar corrupción de estado.

```
//-- Uso directo de valor externo en operación sensible
const amount = req.body.amount;
account.balance -= amount;
```

```
//-- Conversión explícita y validación de rango permitido
const amount = Number(req.body.amount);
if (!Number.isFinite(amount) || amount <= 0 || amount > 10000) { throw new
Error('Invalid amount'); }
account.balance -= amount;
```

- Definir listas blancas para valores esperados en lugar de confiar en cualquier valor recibido.

```
//-- Uso directo de parámetro para determinar rol
const role = req.body.role;
user.role = role;
```

```
//-- Validación contra lista blanca de valores permitidos
const allowedRoles = ['user', 'admin'];
const role = String(req.body.role);
if (!allowedRoles.includes(role)) { throw new Error('Invalid role'); }
user.role = role;
```

2.3.2.2 Estándares metodológicos

- **JS-WASCP-2.1**: Validation.
- **JS-WASCP-2.2**: Sanitization.
- **JS-WASCP-3.1.3**: How to prevent.

2.3.3 [js.val.3] - Evitar el uso de APIs obsoletas o inseguras

ID	js.val.3	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las APIs obsoletas presentan **vulnerabilidades conocidas** y comportamientos impredecibles que comprometen la seguridad de las aplicaciones. Muchas funciones deprecadas carecen de validaciones adecuadas, permiten inyección de código o exponen datos sensibles. Su uso continuado introduce **riesgos de seguridad** evitables y dificulta el mantenimiento del código.

Codificación insegura

Uso de una API de Node.js marcada como obsoleta para manipular buffers de memoria. Se busca mostrar cómo una API antigua puede generar comportamientos inseguros o ambiguos, especialmente cuando se trabaja con datos sensibles, y por qué es importante revisar advertencias de deprecación.

```
//-- Uso de la API obsoleta new Buffer() que puede provocar problemas de seguridad
const data = "Usuario:admin";

//-- Creación de un buffer usando constructor obsoleto
const buffer = new Buffer(data);

//-- Conversión del contenido a base64
const encoded = buffer.toString("base64");

//-- Impresión del resultado en consola
console.log("Contenido codificado:", encoded);

//-- Finalización del proceso
process.exit(0);
```

Codificación recomendada

Uso de la API moderna y soportada para manipular buffers en Node.js. Se pretende mostrar cómo reemplazar APIs obsoletas por alternativas seguras y oficialmente mantenidas, reduciendo riesgos de comportamiento indefinido y mejorando la claridad del código.

```
//-- Uso de la API recomendada Buffer.from() para crear buffers de forma segura
const data = "Usuario:admin";

//-- Creación de un buffer usando la API actual y soportada
const buffer = Buffer.from(data, "utf8");

//-- Conversión del contenido a base64
const encoded = buffer.toString("base64");

//-- Impresión del resultado en consola
console.log("Contenido codificado:", encoded);

//-- Finalización del proceso
process.exit(0);
```

2.3.3.1 Descripción

Las APIs obsoletas representan funcionalidades que han sido marcadas para eliminación o reemplazo debido a problemas de seguridad, rendimiento o diseño. Estas interfaces mantienen compatibilidad temporal, pero introducen vulnerabilidades conocidas y comportamientos impredecibles en el código.

El uso de APIs obsoletas genera **comportamientos indefinidos** que pueden manifestarse de forma inconsistente entre diferentes entornos de ejecución, versiones del motor o

configuraciones del sistema. Esta inconsistencia compromete la predictibilidad del código y puede exponer vectores de ataque no documentados.

Las APIs inseguras incluyen funciones que procesan datos sin validación adecuada, manejan memoria de forma incorrecta o implementan algoritmos criptográficos débiles. Su utilización puede resultar en **exposición de datos sensibles**, corrupción de memoria o ejecución de código malicioso.

Los motores modernos pueden modificar o eliminar el soporte para estas APIs sin previo aviso, causando fallos en tiempo de ejecución. La dependencia de funcionalidades obsoletas también limita las opciones de actualización y mantenimiento del código.

La mitigación requiere **auditorías regulares** del código para identificar el uso de APIs marcadas como obsoletas o inseguras. La sustitución por alternativas modernas debe realizarse siguiendo las recomendaciones oficiales de la especificación del lenguaje y las mejores prácticas de seguridad establecidas.

Las herramientas de análisis estático pueden automatizar la detección de estas APIs problemáticas durante el desarrollo. La implementación de políticas de codificación que prohíban explícitamente su uso previene la introducción de nuevas vulnerabilidades en el código base.

Riesgos asociados

- Ejecución de código inesperado debido a comportamientos no documentados o indefinidos.
- Introducción de vulnerabilidades conocidas por la comunidad y los atacantes.
- Falta de soporte para futuras actualizaciones de navegadores o entornos de ejecución.
- Posibilidad de explotación por ataques de tipo XSS o CSRF por manejo deficiente de datos.
- Pérdida de interoperabilidad con bibliotecas o frameworks modernos.
- Dificultad para aplicar parches de seguridad o recibir actualizaciones del proveedor.
- Mala gestión de la memoria, provocando fugas (memory leaks) o fallos de la aplicación.
- Generación de errores silenciosos difíciles de detectar y depurar.
- Incompatibilidad con sistemas de análisis estático y herramientas modernas de desarrollo seguro.
- Desviaciones del estándar ECMAScript, dificultando la portabilidad y mantenibilidad del código.

Buenas prácticas

- Utilizar APIs modernas y mantenidas en lugar de funciones obsoletas o con comportamientos inconsistentes.

```
//-- Uso de función obsoleta para decodificación
var texto = unescape(cadenaCodificada);
```

```
//-- Uso de API estándar y mantenida
var texto = decodeURIComponent(cadenaCodificada);
```

- Reemplazar métodos criptográficos inseguros u obsoletos por APIs criptográficas actuales y robustas.

```
//-- Uso de algoritmo hash obsoleto
const crypto = require('crypto');
  const hash = crypto.createHash('md5').update(password).digest('hex');
```

```
//-- Uso de algoritmo hash robusto
const crypto = require('crypto');
  const hash = crypto.createHash('sha256').update(password).digest('hex');
```

- Evitar temporizadores con evaluación implícita de código y usar funciones explícitas.

```
//-- Temporizador con evaluación de cadena
setTimeout("ejecutarProceso()", 1000);
```

```
//-- Temporizador con referencia directa a función
setTimeout(ejecutarProceso, 1000);
```

- Sustituir APIs síncronas bloqueantes obsoletas por alternativas asíncronas modernas.

```
//-- Lectura síncrona que bloquea el event loop
const fs = require('fs');
  const datos = fs.readFileSync('archivo.txt', 'utf8');
```

```
//-- Lectura asíncrona no bloqueante
const fs = require('fs').promises;
  fs.readFile('archivo.txt', 'utf8')
    .then(datos => procesar(datos));
```

- Evitar propiedades o APIs marcadas como deprecated y utilizar sus reemplazos

oficiales.

```
//-- Uso de propiedad obsoleta
var ancho = window.innerWidth || document.body.clientWidth;
```

```
//-- Uso de propiedad estándar recomendada
var ancho = document.documentElement.clientWidth;
```

2.3.3.2 Estándares metodológicos

- **JS-WASCP-14.1**: Dependencies.
- **Airbnb-JS-29**: Standard Library.
- **MDN Web Docs JS-2**: Grammar and types.

2.4 Gestión de errores

2.4.1 [js.err.1] - Capturar y gestionar todas las excepciones

ID	js.err.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las excepciones no capturadas provocan terminaciones abruptas del programa y pueden exponer información sensible del sistema. El manejo inadecuado de errores genera **vulnerabilidades de seguridad** y comportamientos impredecibles que comprometen la estabilidad de la aplicación.

Codificación insegura

Lectura de un archivo JSON desde disco para procesar su contenido. Se busca mostrar cómo se gestiona un posible error durante la lectura o el parseo, y qué ocurre cuando la excepción se captura, pero no se trata de forma explícita.

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

function leerConfiguracion(ruta) {
  //-- Intenta leer y parsear un archivo JSON
  try {
    const contenido = fs.readFileSync(ruta, 'utf8');
    const config = JSON.parse(contenido);
    return config;
  } catch (error) {
    //-- Se captura la excepción, pero no se hace nada con ella
  }
}
```

```
function iniciarAplicacion() {
  //-- Se intenta usar la configuración sin validar si hubo error
  const config = leerConfiguracion('config.json');
  console.log('Puerto configurado:', config.puerto);
}
iniciarAplicacion();
```

Codificación recomendada

Lectura de un archivo JSON desde disco para procesar su contenido, gestionando de forma explícita los posibles errores. Se muestra cómo registrar el problema, detener la ejecución de forma controlada o propagar la excepción para evitar estados inconsistentes.

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

function leerConfiguracion(ruta) {
  //-- Intenta leer y parsear un archivo JSON
  try {
    const contenido = fs.readFileSync(ruta, 'utf8');
    const config = JSON.parse(contenido);
    return config;
  } catch (error) {
    //-- Se registra el error para auditoría y diagnóstico
    console.error('Error al leer la configuración:', error.message);

    //-- Se propaga la excepción para que el flujo superior decida cómo actuar
    throw error;
  }
}
function iniciarAplicacion() {
  //-- Se controla el error en el punto de entrada de la aplicación
  try {
    const config = leerConfiguracion('config.json');
    console.log('Puerto configurado:', config.puerto);
  } catch (error) {
    //-- Se finaliza la aplicación de forma controlada
    console.error('No se pudo iniciar la aplicación. Terminando proceso.');
    process.exit(1);
  }
}
iniciarAplicacion();
```

2.4.1.1 Descripción

El manejo explícito de excepciones constituye un mecanismo de control de flujo que permite interceptar y procesar errores durante la ejecución del programa. Esta práctica

requiere el uso de estructuras `try-catch` para capturar excepciones y definir acciones específicas para cada tipo de error.

La omisión del manejo de excepciones puede provocar **terminaciones inesperadas del programa**, exposición de información sensible a través de mensajes de error del sistema, y comportamientos impredecibles que comprometen la estabilidad de la aplicación. Los errores no controlados pueden propagarse hasta niveles superiores del sistema, afectando componentes no relacionados con el origen del problema.

Las operaciones asíncronas presentan riesgos adicionales cuando no se implementa un manejo adecuado de errores. Las promesas rechazadas sin captura pueden generar **advertencias de rechazo no manejado** y potencialmente causar fallos en el proceso de ejecución.

La mitigación efectiva implica la implementación sistemática de bloques de captura de errores en todas las operaciones que puedan fallar. Esto incluye el uso de `catch` en promesas, manejo de errores en funciones asíncronas con `async/await`, y la definición de estrategias de recuperación apropiadas para cada contexto.

El registro detallado de errores facilita el diagnóstico y resolución de problemas, mientras que la implementación de **mecanismos de fallback** permite mantener la funcionalidad básica del sistema ante fallos parciales. La validación de entrada y la verificación de precondiciones reducen la probabilidad de excepciones durante la ejecución.

Riesgos asociados

- Pérdida de trazabilidad de fallos, dificultando la identificación y resolución de errores.
- Comportamiento inesperado de la aplicación debido a errores no controlados.
- Fugas de información sensible si los errores no se manejan de forma segura y se exponen al usuario final.
- Interrupción del flujo normal de la aplicación, afectando la disponibilidad o la experiencia del usuario.
- Ataques de denegación de servicio (DoS) derivados de fallos repetidos no gestionados.
- Fallo en la liberación de recursos, causando pérdidas de memoria o bloqueos.
- Exposición accidental de detalles internos de la aplicación, facilitando ataques dirigidos.
- Corrupción o pérdida de datos causada por errores no capturados durante operaciones críticas.
- Incremento en la dificultad del mantenimiento y la depuración del código.

Buenas prácticas

- Registrar y manejar explícitamente las excepciones en bloques `try/catch`.

```
//-- Captura vacía que ignora el error
try {
  procesarPago(datos);
} catch (e) {
}
```

```
//-- Captura con registro y acción de manejo
try {
  procesarPago(datos);
} catch (e) {
  logger.error('Error al procesar pago', e);
  notificarAdministrador(e);
  throw e;
}
```

- Propagar la excepción cuando no se puede manejar adecuadamente en el contexto actual.

```
//-- Se oculta el error devolviendo un valor por defecto
function obtenerUsuario(id) {
  try {
    return buscarUsuario(id);
  } catch (e) {
    return null;
  }
}
```

```
//-- Se propaga el error para que sea tratado en una capa superior
function obtenerUsuario(id) {
  try {
    return buscarUsuario(id);
  } catch (e) {
    logger.warn('Fallo al obtener usuario', e);
    throw new Error('No se pudo obtener el usuario');
  }
}
```

- Manejar explícitamente las promesas rechazadas usando `async/await` con `try/catch` o `.catch()`.

```
//-- Promesa sin manejo de rechazo
obtenerDatosRemotos()
  .then(data => {
```

```
    procesar(data);
  });
```

```
//-- Promesa con manejo explícito del error
obtenerDatosRemotos()
  .then(data => {
    procesar(data);
  })
  .catch(error => {
    logger.error('Error en datos remotos', error);
    alertarSoporte(error);
  });
```

2.4.1.2 Estándares metodológicos

- **JS-WASCP-8.1**: Error Handling.
- **MDN Web Docs JS-4**: Control flow and error handling.

2.4.2 [js.err.2] - Evitar lanzar valores no-Error como excepciones

ID	js.err.2	Uso	Recomendado
CWSS Score	62	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

El lanzamiento de valores que no son instancias de `Error` como excepciones compromete la **semántica del manejo de errores** y dificulta la trazabilidad durante la depuración. Esta práctica genera inconsistencias en el tratamiento de excepciones y reduce la información disponible para diagnosticar problemas en tiempo de ejecución.

Codificación insegura

Gestión de errores en una función que valida la edad de un usuario antes de procesar una operación. Se busca mostrar cómo la forma de lanzar excepciones impacta en la trazabilidad, el stack trace y la capacidad de manejo uniforme de errores.

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

//-- Función que valida la edad y lanza valores primitivos como excepción
function validarEdad(edad) {
  //-- Se lanza un string en lugar de un objeto Error
  if (edad === undefined) {
    throw "Edad no proporcionada";
  }
  //-- Se lanza un número como excepción
  if (isNaN(edad)) {
```

```
    throw 400;
  }
  if (edad < 18)
    throw "Usuario menor de edad";
  }
  return "Acceso concedido";
}
//-- Ejecución principal con manejo genérico de errores
try {
  const resultado = validarEdad("abc");
  console.log(resultado);
} catch (error) {
  //-- Se pierde información de trazabilidad y stack trace estructurado
  console.error("Error capturado:", error);
}
```

Codificación recomendada

Gestión de errores en la misma validación de edad, utilizando objetos derivados de `Error` para mantener semántica clara, stack trace consistente y facilitar el manejo específico de excepciones.

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

//-- Definición de un error personalizado que extiende de Error
class ValidacionError extends Error {
  constructor(mensaje) {
    super(mensaje);
    this.name = "ValidacionError";
  }
}
//-- Función que valida la edad y lanza objetos Error
function validarEdad(edad) {
  if (edad === undefined) {
    throw new ValidacionError("Edad no proporcionada");
  }
  if (isNaN(edad)) {
    throw new ValidacionError("Edad debe ser numérica");
  }
  if (edad < 18) {
    throw new ValidacionError("Usuario menor de edad");
  }
  return "Acceso concedido";
}
//-- Ejecución principal con manejo específico y trazabilidad completa
try {
  const resultado = validarEdad("abc");
  console.log(resultado);
```

```
} catch (error) {
  //-- Se mantiene stack trace y tipado consistente
  if (error instanceof ValidacionError) {
    console.error("Error de validación:", error.message);
    console.error("Stack trace:", error.stack);
  } else {
    console.error("Error inesperado:", error);
  }
}
```

2.4.2.1 Descripción

El mecanismo de excepciones permite lanzar cualquier tipo de valor mediante la instrucción `throw`, incluyendo cadenas de texto, números, objetos literales o valores primitivos. Sin embargo, esta flexibilidad puede comprometer la consistencia del manejo de errores y la capacidad de diagnóstico del sistema.

Cuando se lanzan valores que no son instancias de `Error` o sus subclases, se pierde información crítica para el diagnóstico. Los objetos `Error` proporcionan automáticamente propiedades como `stack`, `name` y `message` que facilitan la identificación del origen del problema y su contexto de ejecución.

La inconsistencia en los tipos de excepciones lanzadas complica el manejo uniforme de errores. Los bloques `catch` deben implementar lógica adicional para determinar el tipo de valor recibido y procesarlo adecuadamente, lo que incrementa la complejidad del código y la probabilidad de errores.

Esta práctica también afecta a la integración con herramientas de monitoreo y logging, que esperan recibir objetos con estructura estándar para extraer información relevante. Los valores primitivos u objetos arbitrarios pueden no contener los datos necesarios para un análisis efectivo de los fallos.

La mitigación de estos problemas requiere establecer una política consistente para el lanzamiento de excepciones, utilizando únicamente objetos que hereden de `Error`. Esto incluye tanto la clase base como sus especializaciones nativas o personalizadas que mantengan la estructura y comportamiento esperados.

Las herramientas de análisis estático pueden configurarse para detectar usos de `throw` con valores no conformes, mientras que las revisiones de código deben verificar el cumplimiento de esta práctica en todo el sistema.

Riesgos asociados

- Pérdida de información esencial sobre el origen del error debido a la ausencia de stack trace adecuado.
- Dificultad para depurar y rastrear problemas, complicando el mantenimiento y la resolución de incidentes.

- Posibilidad de que los manejadores de errores no identifiquen o ignoren excepciones lanzadas, al no ser instancias de `Error`.
- Ruptura de herramientas de supervisión y registro que esperan objetos `Error` o estructuras estándar.
- Comportamientos inconsistentes en entornos de ejecución y frameworks que dependen del tipo `Error` para el tratamiento correcto de excepciones.
- Exposición accidental de información sensible si se lanzan valores inadecuados o datos de usuario como excepción.
- Mayor riesgo de ocultar vulnerabilidades de seguridad, ya que los errores lanzados informalmente pueden pasar desapercibidos.

Buenas prácticas

- Lanzar siempre instancias de `Error` o clases que hereden de `Error` en lugar de valores primitivos.

```
//-- Se lanza un valor primitivo sin información estructurada
if (!user) {
  throw "Usuario no encontrado";
}
```

```
//-- Se lanza un objeto Error con mensaje descriptivo
if (!user) {
  throw new Error("Usuario no encontrado");
}
```

- Crear clases de error personalizadas que extiendan `Error` para mantener semántica y trazabilidad.

```
//-- Se lanza un objeto genérico sin contexto específico
if (!hasAccess) {
  throw new Error("Acceso denegado");
}
```

```
//-- Se define una clase específica que extiende Error
class AuthorizationError extends Error {
  constructor(message) {
    super(message);
    this.name = "AuthorizationError";
  }
}
if (!hasAccess) {
  throw new AuthorizationError("Acceso denegado");
```

```
}
```

- Incluir información contextual relevante dentro del objeto `Error` en lugar de lanzar estructuras arbitrarias.

```
//-- Se lanza un objeto literal sin trazabilidad estándar
if (!fileExists) {
  throw { code: 404, message: "Archivo no encontrado" };
}
```

```
//-- Se adjunta información contextual a una instancia de Error
if (!fileExists) {
  const err = new Error("Archivo no encontrado");
  err.code = 404;
  throw err;
}
```

- Preservar la causa original del error utilizando la propiedad `cause` al relanzar excepciones.

```
//-- Se relanza un valor sin mantener la causa original
try {
  processFile();
} catch (e) {
  throw "Error procesando archivo";
}
```

```
//-- Se encapsula el error original preservando la causa
try {
  processFile();
} catch (e) {
  throw new Error("Error procesando archivo", { cause: e });
}
```

2.4.2.2 Estándares metodológicos

- **JS-WASCP-8.1**: Error Handling.
- **Google-JS-5.8**: Control structures.

2.4.3 [js.err.3] - Evitar la ejecución de errores silenciosos

ID	js.err.3	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Los errores silenciosos en tiempo de ejecución pueden pasar desapercibidos durante el desarrollo y causar comportamientos inesperados en producción. El motor de JavaScript permite que muchas operaciones problemáticas se ejecuten sin generar excepciones, como asignaciones a propiedades de objetos no extensibles o el uso de variables no declaradas. Esta tolerancia puede enmascarar **defectos críticos** que comprometen la funcionalidad y seguridad del código.

Codificación insegura

Se muestra un script que ejecuta operaciones potencialmente problemáticas sin activar modo estricto ni manejar adecuadamente los errores en tiempo de ejecución. La intención es observar cómo ciertos fallos, como el uso de variables no declaradas o promesas rechazadas, pueden pasar desapercibidos o generar comportamientos inesperados.

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

//-- función que usa una variable no declarada
function calcularTotal(precio, cantidad) {
  total = precio * cantidad;
  return total;
}
//-- promesa que se rechaza sin manejo adecuado
function leerArchivo(ruta) {
  return new Promise((resolve, reject) => {
    fs.readFile(ruta, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}
//-- ejecución principal sin control de errores
const resultado = calcularTotal(10, 5);
console.log('Resultado:', resultado);

leerArchivo('archivo_inexistente.txt');
```

Codificación recomendada

Se presenta un script configurado para detectar fallos en tiempo de ejecución mediante el uso de modo estricto y manejadores explícitos de errores. El objetivo es asegurar que problemas como variables no declaradas o promesas rechazadas generen fallos controlados y visibles, facilitando su detección y corrección.

```
'use strict';

const fs = require('fs');

//-- manejo global de errores no controlados
process.on('uncaughtException', (err) => {
  console.error('Excepción no controlada:', err.message);
  process.exit(1);
});

//-- manejo global de promesas rechazadas
process.on('unhandledRejection', (reason) => {
  console.error('Promesa rechazada no controlada:', reason.message);
  process.exit(1);
});

//-- función que declara correctamente sus variables
function calcularTotal(precio, cantidad) {
  const total = precio * cantidad;
  return total;
}
//-- función que propaga errores de forma controlada
function leerArchivo(ruta) {
  return fs.promises.readFile(ruta, 'utf8');
}
//-- ejecución principal con manejo explícito de errores
(async () => {
  try {
    const resultado = calcularTotal(10, 5);
    console.log('Resultado:', resultado);

    const contenido = await leerArchivo('archivo_inexistente.txt');
    console.log(contenido);
  } catch (err) {
    console.error('Error controlado:', err.message);
    process.exit(1);
  }
})();
```

2.4.3.1 Descripción

Los errores silenciosos representan fallos que ocurren durante la ejecución del programa, pero no generan excepciones ni interrumpen el flujo normal de ejecución. Estos errores pueden manifestarse como operaciones que fallan sin notificación, asignaciones a propiedades inexistentes, o llamadas a métodos sobre valores `undefined` que retornan resultados inesperados.

La detección temprana de estos fallos resulta crítica para mantener la **integridad del programa** y prevenir comportamientos impredecibles. Los errores no detectados pueden propagarse a través del sistema, causando **corrupción de datos**, lógica incorrecta o fallos en cascada que son difíciles de rastrear hasta su origen.

El modo estricto constituye el mecanismo principal para convertir errores silenciosos en excepciones explícitas. Al activar `strict mode`, el intérprete genera errores para operaciones que normalmente fallarían de forma silenciosa, como asignaciones a variables no declaradas, eliminación de propiedades no configurables, o uso de nombres de parámetros duplicados.

La configuración del entorno de desarrollo debe incluir herramientas de análisis estático que identifiquen patrones propensos a errores silenciosos. Los linters y analizadores de código pueden detectar construcciones problemáticas antes de la ejecución, mientras que las herramientas de debugging avanzadas permiten monitorear el comportamiento del programa en tiempo real.

La mitigación efectiva requiere una combinación de configuración del entorno de ejecución, uso de herramientas de análisis e implementación de mecanismos de validación explícitos. La adopción de prácticas de programación defensiva, junto con la configuración adecuada del intérprete, reduce significativamente la probabilidad de que los errores pasen desapercibidos durante el desarrollo y la producción.

Riesgos asociados

- Difícil identificación y diagnóstico de bugs o vulnerabilidades en el código.
- Exposición a fallos de seguridad que puedan ser explotados por atacantes (por ejemplo, ejecución de código no previsto o acceso no autorizado).
- Comportamientos inesperados de la aplicación sin notificación ni registro de los mismos.
- Pérdida de integridad de datos debido al manejo incorrecto de errores.
- Desviaciones en los flujos de negocio sin que el equipo de desarrollo o los usuarios lo detecten.
- Incremento de los costes de mantenimiento y soporte por dificultad en la trazabilidad de problemas.
- Falta de cumplimiento de normativas de seguridad que exigen el registro y

gestión de errores.

- Riesgo de denegaciones de servicio (DoS) si los errores no manejados correctamente afectan la estabilidad de la aplicación.
- Posibilidad de filtrado de información sensible si los errores no se gestionan y ocultan apropiadamente.

Buenas prácticas

- Activar el modo estricto para evitar errores silenciosos y comportamientos implícitos inseguros.

```
//-- Variable no declarada genera variable global implícita
function calcular() {
  total = 100;
  return total;
}
```

```
//-- Modo estricto detecta asignaciones a variables no declaradas
"use strict";
function calcular() {
  let total = 100;
  return total;
}
```

- Configurar linters (por ejemplo, ESLint) para detectar errores potenciales en tiempo de desarrollo.

```
//-- Código con variable redefinida y no utilizada
function procesar(datos) {
  var resultado = datos;
  var resultado = 10;
  return resultado;
}
```

```
//-- Uso de reglas estrictas que evitan redeclaraciones y variables sin uso
/* eslint no-redeclare: "error", no-unused-vars: "error" */
function procesar(datos) {
  let resultado = Number(datos);
  return resultado;
}
```

- Gestionar explícitamente las promesas para evitar fallos silenciosos en operaciones asíncronas.

```
//-- Promesa sin manejo de error puede ocultar fallos
```

```
fetch(url)
  .then(res => res.json());
```

```
//-- Manejo explícito de errores en la cadena de promesas
fetch(url)
  .then(res => {
    if (!res.ok) throw new Error("Error HTTP");
    return res.json();
  })
  .catch(err => {
    console.error("Fallo controlado:", err);
  });
```

- Configurar Node.js para finalizar el proceso ante excepciones no controladas.

```
//-- Excepción no capturada que puede dejar la aplicación en estado inconsistente
setTimeout(() => {
  throw new Error("Fallo inesperado");
}, 100);
```

```
//-- Gestión explícita de excepciones no capturadas
process.on("uncaughtException", err => {
  console.error("Error crítico:", err);
  process.exit(1);
});
setTimeout(() => {
  throw new Error("Fallo inesperado");
}, 100);
```

- Validar explícitamente los valores devueltos para evitar suposiciones implícitas que oculten errores.

```
//-- Suposición implícita de que el valor siempre es válido
let edad = parseInt(input);
if (edad >= 18) {
  acceder();
}
```

```
//-- Validación explícita del resultado antes de usarlo
let edad = parseInt(input, 10);
if (Number.isNaN(edad)) {
  throw new Error("Edad inválida");
}
if (edad >= 18) {
  acceder();
```

```
}
```

2.4.3.2 Estándares metodológicos

- **JS-WASCP-8.1**: Error Handling.
- **Airbnb-JS-7**: Functions.

2.5 Concurrencia

2.5.1 [js.sync.1] - Gestionar errores en promesas mediante try/catch con async/await

ID	js.sync.1	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las promesas no gestionadas pueden provocar errores silenciosos que comprometen la estabilidad de la aplicación. Cuando las operaciones asíncronas fallan sin un manejo explícito de errores, el programa puede continuar ejecutándose en un **estado inconsistente**, generando comportamientos impredecibles y dificultando la detección de fallos en producción.

Codificación insegura

Gestión de una operación asíncrona que simula la consulta a un servicio externo sin manejar correctamente los errores. Se busca observar cómo una promesa rechazada puede generar estados inconsistentes o finalizar la aplicación de forma inesperada cuando no se controla la excepción.

```
const obtenerUsuario = () => {
  //-- Simula una llamada a un servicio externo que falla
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(new Error("Error al obtener el usuario"));
    }, 1000);
  });
};

const procesarUsuario = async () => {
  //-- Llama a la función asincrónica sin manejar posibles errores
  const usuario = await obtenerUsuario();

  //-- Esta línea no se ejecutará si ocurre un error
  console.log("Usuario obtenido:", usuario);
};
```

```
procesarUsuario();
```

Codificación recomendada

Gestión de la misma operación asíncrona controlando explícitamente los errores mediante `try/catch` con `async/await`. Se busca garantizar un flujo estable, registrar el problema y evitar que la aplicación quede en un estado inconsistente.

```
const obtenerUsuario = () => {
  //-- Simula una llamada a un servicio externo que falla
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(new Error("Error al obtener el usuario"));
    }, 1000);
  });
};

const procesarUsuario = async () => {
  try {
    //-- Llama a la función asincrónica y controla posibles errores
    const usuario = await obtenerUsuario();

    //-- Solo se ejecuta si la promesa se resuelve correctamente
    console.log("Usuario obtenido:", usuario);
  } catch (error) {
    //-- Manejo explícito del error para evitar estados inconsistentes
    console.error("Se produjo un error controlado:", error.message);

    //-- Aquí podrían aplicarse acciones compensatorias o de recuperación
  }
};

procesarUsuario();
```

2.5.1.1 Descripción

Las promesas representan operaciones asíncronas que pueden completarse exitosamente o fallar. Cuando no se gestionan explícitamente los errores mediante `catch` o bloques `try/catch` con `async/await`, las excepciones no capturadas pueden propagarse de forma impredecible a través de la aplicación.

Los errores no manejados en promesas generan **unhandled promise rejections**, que pueden provocar terminaciones inesperadas del proceso o dejar la aplicación en estados inconsistentes. Estos fallos silenciosos dificultan la detección de problemas durante el desarrollo y pueden manifestarse como comportamientos erráticos en producción.

La ausencia de manejo explícito de errores impide que la aplicación responda adecuadamente a condiciones de fallo, como problemas de conectividad, recursos no disponibles o datos malformados. Esta situación compromete la **estabilidad del sistema** y la experiencia del usuario.

El manejo explícito de errores permite implementar estrategias de recuperación apropiadas, como reintentos automáticos, valores por defecto o notificaciones al usuario. La captura sistemática de excepciones facilita el registro de errores para diagnóstico y mejora la capacidad de monitoreo de la aplicación.

Los entornos de ejecución modernos pueden configurarse para tratar las promesas rechazadas no manejadas como errores fatales, terminando el proceso automáticamente. Esta configuración refuerza la importancia de implementar manejo de errores exhaustivo en todas las operaciones asíncronas.

Riesgos asociados

- Pérdida de trazabilidad de errores que pueden pasar desapercibidos (errores silenciosos).
- Estados inconsistentes en la aplicación debido a operaciones incompletas o mal gestionadas.
- Filtración de datos sensibles si la aplicación falla inesperadamente y expone información por falta de manejo adecuado de errores.
- Vulnerabilidad a ataques de denegación de servicio (DoS) si un error no gestionado detiene ejecuciones clave o bloquea recursos.
- Corrupción de datos al no poder garantizar la finalización o el rollback de procesos asíncronos fallidos.
- Degradación en la experiencia del usuario debido a la ausencia de retroalimentación clara cuando ocurre un error.
- Imposibilidad de implementar medidas de recuperación o remediación tras un fallo, facilitando la explotación repetida del mismo fallo.

Buenas prácticas

- Gestionar siempre los errores en promesas utilizando `.catch()` para evitar estados inconsistentes o fallos silenciosos.

```
//-- Promesa sin gestión de errores
fetch('/api/datos')
  .then(response => response.json())
  .then(data => {
    procesarDatos(data);
  });
```

```
//-- Promesa con manejo explícito de errores
fetch('/api/datos')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la respuesta');
    }
    return response.json();
  })
  .then(data => {
    procesarDatos(data);
  })
  .catch(error => {
    registrarError(error);
  });
```

- Utilizar bloques `try/catch` al trabajar con `async/await` para capturar excepciones y mantener la aplicación en un estado controlado.

```
//-- Uso de async/await sin captura de errores
async function obtenerDatos() {
  const response = await fetch('/api/datos');
  const data = await response.json();
  procesarDatos(data);
}
```

```
//-- Uso de async/await con control de errores
async function obtenerDatos() {
  try {
    const response = await fetch('/api/datos');
    if (!response.ok) {
      throw new Error('Error en la respuesta');
    }
    const data = await response.json();
    procesarDatos(data);
  } catch (error) {
    registrarError(error);
  }
}
```

- Propagar los errores cuando no puedan manejarse localmente, asegurando que capas superiores puedan gestionarlos adecuadamente.

```
//-- Error capturado, pero no propagado
function obtenerUsuario() {
  return fetch('/api/usuario')
    .then(res => res.json())
    .catch(error => {
```

```
      registrarError(error);
    });
}
```

```
//-- Error registrado y propagado correctamente
function obtenerUsuario() {
  return fetch('/api/usuario')
    .then(res => res.json())
    .catch(error => {
      registrarError(error);
      throw error;
    });
}
```

2.5.1.2 Estándares metodológicos

- **JS-WASCP-8.1**: Error Handling.
- **MDN Web Docs JS-15**: Promises.

2.5.2 [js.sync.2] - Proteger el acceso a recursos compartidos en código asíncrono

ID	js.sync.2	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las operaciones asíncronas pueden generar **condiciones de carrera** cuando múltiples funciones acceden simultáneamente a los mismos recursos compartidos. Estos accesos concurrentes producen comportamientos impredecibles y resultados inconsistentes, especialmente en aplicaciones que manejan estados globales, variables compartidas o recursos externos como archivos y bases de datos.

Codificación insegura

Se muestra una situación donde varias operaciones asíncronas modifican un recurso compartido sin ningún mecanismo de sincronización. El objetivo es entender cómo las operaciones concurrentes pueden provocar resultados inconsistentes cuando no se controla el acceso al estado compartido.

```
//-- Recurso compartido
let contador = 0;

//-- Función asíncrona que incrementa el contador
async function incrementar() {
  //-- Simula una operación asíncrona
```

```
  const valorActual = contador;
  await new Promise(resolve => setTimeout(resolve, 100));
  contador = valorActual + 1;
}
async function ejecutar() {
  //-- Se lanzan varias operaciones en paralelo
  await Promise.all([
    incrementar(),
    incrementar(),
    incrementar()
  ]);

  //-- Resultado inesperado por condición de carrera
  console.log('Valor final del contador:', contador);
}
//-- Punto de entrada
ejecutar();
```

Codificación recomendada

Se presenta el mismo escenario de incremento concurrente sobre un recurso compartido, incorporando un mecanismo de exclusión mutua para serializar el acceso. La intención es mostrar cómo proteger el estado compartido en código asíncrono para garantizar resultados consistentes.

```
'use strict';

//-- Implementación simple de un mutex
class Mutex {
  constructor() {
    this.locked = false;
    this.queue = [];
  }
  lock() {
    return new Promise(resolve => {
      if (!this.locked) {
        this.locked = true;
        resolve();
      } else {
        this.queue.push(resolve);
      }
    });
  }
  unlock() {
    if (this.queue.length > 0) {
      const nextResolve = this.queue.shift();
      nextResolve();
    } else {
```

```
      this.locked = false;
    }
  }
}
//-- Recurso compartido
let contador = 0;
const mutex = new Mutex();

//-- Función asíncrona protegida por mutex
async function incrementar() {
  await mutex.lock();
  try {
    //-- Sección crítica protegida
    const valorActual = contador;
    await new Promise(resolve => setTimeout(resolve, 100));
    contador = valorActual + 1;
  } finally {
    //-- Liberación del recurso
    mutex.unlock();
  }
}
async function ejecutar() {
  //-- Operaciones en paralelo con acceso controlado
  await Promise.all([
    incrementar(),
    incrementar(),
    incrementar()
  ]);

  //-- Resultado consistente
  console.log('Valor final del contador:', contador);
}
//-- Punto de entrada
ejecutar();
```

2.5.2.1 Descripción

Las condiciones de carrera ocurren cuando múltiples operaciones asíncronas acceden simultáneamente a recursos compartidos sin coordinación adecuada. Este problema surge frecuentemente en aplicaciones que manejan operaciones concurrentes, como peticiones HTTP, acceso a bases de datos o manipulación del DOM desde diferentes contextos de ejecución.

El modelo de ejecución asíncrono permite que múltiples operaciones se ejecuten de forma no bloqueante, pero introduce complejidad en el manejo de estados compartidos. Cuando dos o más operaciones asíncronas modifican el mismo recurso sin sincronización, el resultado final depende del orden de ejecución, generando comportamientos impredecibles.

Los recursos compartidos incluyen variables globales, propiedades de objetos, elementos del DOM, conexiones de red y cualquier estado que persista entre diferentes ciclos de ejecución. El acceso no coordinado a estos recursos puede corromper datos, generar inconsistencias en el estado de la aplicación y provocar fallos difíciles de reproducir y diagnosticar.

La protección de recursos compartidos requiere implementar mecanismos de **sincronización** que coordinen el acceso entre operaciones concurrentes. Estos mecanismos incluyen el uso de promesas encadenadas, funciones `async/await` para serializar operaciones críticas, y patrones de diseño que eviten el acceso simultáneo a estados mutables.

La mitigación efectiva implica identificar puntos de acceso concurrente, implementar estrategias de sincronización apropiadas y diseñar arquitecturas que minimicen la dependencia de estados compartidos mutables. El uso de estructuras de datos inmutables y patrones funcionales reduce significativamente la superficie de ataque para este tipo de vulnerabilidades.

Riesgos asociados

- Corrupción de datos debido a actualizaciones no sincronizadas de recursos compartidos.
- Errores lógicos impredecibles que pueden ser muy difíciles de identificar y depurar.
- Pérdida o sobrescritura de información crítica por acceso concurrente no controlado.
- Incremento de vulnerabilidades de seguridad, como ataques de tipo "race condition exploitation".
- Desincronización en el flujo de ejecución, generando respuestas incorrectas en aplicaciones web o API.
- Caídas o bloqueos inesperados de la aplicación por recursos en estados inconsistentes.
- Fugas de datos sensibles si condiciones de carrera permiten el acceso no autorizado.
- Dificultad para garantizar la integridad y consistencia de la información a lo largo de la aplicación.

Buenas prácticas

- Sincronizar el acceso a recursos compartidos utilizando mecanismos de exclusión mutua en operaciones asíncronas.

```
//-- Incremento compartido sin control de concurrencia
let contador = 0;
async function incrementar() {
  let temp = contador;
  await esperar();
  contador = temp + 1;
}
```

```
//-- Incremento protegido mediante bloqueo simple
let contadorSeguro = 0;
let bloqueado = false;
async function incrementarSeguro() {
  while (bloqueado) { await esperar(); }
  bloqueado = true;
  let temp = contadorSeguro;
  await esperar();
  contadorSeguro = temp + 1;
  bloqueado = false;
}
```

- Evitar modificar variables globales desde múltiples funciones asíncronas; encapsular el estado en estructuras controladas.

```
//-- Uso de variable global compartida en múltiples llamadas
let saldo = 100;
async function retirar(monto) {
  if (saldo >= monto) {
    await procesar();
    saldo -= monto;
  }
}
```

```
//-- Encapsulamiento del estado con método controlado
class Cuenta {
  constructor() { this.saldo = 100; }
  async retirar(monto) {
    if (this.saldo >= monto) {
      await procesar();
      this.saldo -= monto;
    }
  }
}
```

- Utilizar estructuras atómicas o colas de tareas para serializar operaciones críticas.

```
//-- Ejecución concurrente sin control del orden
```

```
async function actualizarRegistro() {
  await guardarEnBD();
}
Promise.all([actualizarRegistro(), actualizarRegistro()]);
```

```
//-- Serialización de operaciones mediante cola de promesas
let cola = Promise.resolve();
function actualizarRegistroSeguro() {
  cola = cola.then(() => guardarEnBD());
  return cola;
}
actualizarRegistroSeguro();
actualizarRegistroSeguro();
```

2.5.2.2 Estándares metodológicos

- **JS-WASCP-14.4**: Concurrency.
- **JS-WASCP-14.5**: Sandboxing.

2.5.3 [js.sync.3] - No mezclar callbacks y promesas sin control claro del flujo de ejecución

ID	js.sync.3	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La combinación inadecuada de `callbacks` y **promesas** en el mismo flujo de ejecución genera comportamientos impredecibles y dificulta el control de errores. Esta práctica produce código donde resulta complejo determinar el orden de ejecución y gestionar adecuadamente las excepciones asíncronas.

Codificación insegura

Lectura de un archivo y posterior procesamiento asíncrono combinando `callbacks` y promesas sin un control claro del flujo. Se busca mostrar cómo la mezcla desordenada de ambos mecanismos puede generar ejecuciones inesperadas, manejo incorrecto de errores y dificultad para mantener el código.

```
//-- Importa el módulo de sistema de archivos
const fs = require('fs');

//-- Función que simula una operación asíncrona con promesa
function procesarContenido(contenido) {
  //-- Devuelve una promesa que transforma el texto
  return new Promise((resolve, reject) => {
```

```
    setTimeout(() => {
      if (!contenido) {
        reject(new Error('Contenido vacío'));
      } else {
        resolve(contenido.toUpperCase());
      }
    }, 100);
  });
}
//-- Lectura del archivo usando callback
fs.readFile('datos.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error al leer archivo:', err.message);
    return;
  }
  //-- Se invoca una promesa dentro del callback sin retornar ni encadenar
correctamente
  procesarContenido(data)
    .then(resultado => {
      console.log('Resultado procesado:', resultado);
    });

  //-- Este mensaje puede ejecutarse antes de que termine la promesa
  console.log('Procesamiento lanzado');
});
```

Codificación recomendada

Lectura y procesamiento del mismo archivo utilizando un único modelo asíncrono basado en promesas con `async/await`. Se pretende mostrar un flujo claro, manejo centralizado de errores y mayor mantenibilidad al no mezclar `callbacks` con promesas.

```
'use strict';

//-- Importa la versión de promesas del módulo de sistema de archivos
const fs = require('fs').promises;

//-- Función que simula una operación asíncrona con promesa
function procesarContenido(contenido) {
  //-- Devuelve una promesa que transforma el texto
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!contenido) {
        reject(new Error('Contenido vacío'));
      } else {
        resolve(contenido.toUpperCase());
      }
    }, 100);
```

```
  });
}
//-- Función principal con control claro del flujo asíncrono
async function main() {
  try {
    //-- Lectura del archivo usando promesas
    const data = await fs.readFile('datos.txt', 'utf8');

    //-- Procesamiento encadenado de forma explícita
    const resultado = await procesarContenido(data);

    console.log('Resultado procesado:', resultado);
  } catch (error) {
    //-- Manejo centralizado de errores
    console.error('Error en el flujo:', error.message);
  }
}
//-- Ejecución del flujo principal
main();
```

2.5.3.1 Descripción

El problema surge cuando se utilizan ambos patrones de programación asíncrona sin establecer una estrategia clara de control de flujo. Los `callbacks` siguen un patrón de continuación basado en funciones de retorno, mientras que las promesas implementan un modelo de encadenamiento con métodos `then` y `catch`. La mezcla de estos enfoques sin coordinación adecuada crea puntos de control dispersos y **dificulta la gestión centralizada de errores**.

Los riesgos asociados incluyen condiciones de carrera impredecibles, manejo inconsistente de errores, y dificultades para determinar el estado final de las operaciones asíncronas. La falta de un patrón unificado también complica las tareas de depuración y mantenimiento del código.

La mitigación de estos problemas requiere establecer una **estrategia consistente para el manejo de operaciones asíncronas**. Esto implica seleccionar un patrón principal y convertir las operaciones del patrón secundario al formato del patrón elegido. Las funciones de utilidad para convertir en promesas `callbacks` o para extraer valores de promesas mediante `callbacks` proporcionan mecanismos de conversión controlada entre ambos patrones.

La implementación de un sistema de gestión de errores unificado y el establecimiento de puntos de control claros en el flujo de ejecución contribuyen a mantener la predictibilidad del comportamiento asíncrono. El uso de herramientas de análisis estático también ayuda a identificar patrones mixtos problemáticos durante el desarrollo.

Riesgos asociados

- Condiciones de carrera y fallos impredecibles debido al flujo de ejecución inconsistente.
- Manejo incorrecto de errores, resultando en excepciones no capturadas o silenciosas.
- Ejecución múltiple inadvertida de código (`callbacks` llamados varias veces o promesas resueltas/rechazadas varias veces).
- Dificultad para depurar y mantener el código, aumentando la posibilidad de introducir nuevas vulnerabilidades.
- Fragmentación de la lógica de manejo de errores, facilitando la omisión o el mal tratamiento de fallos de seguridad.
- Riesgo de dejar recursos abiertos o en estado inconsistente si no se gestionan correctamente los flujos asíncronos.
- Exposición a ataques de denegación de servicio (DoS) si el programa entra en bucles asíncronos incontrolados.

Buenas prácticas

- Elegir un único patrón asíncrono (`callbacks` o promesas/`async-await`) y mantener consistencia en todo el flujo.

```
//-- Mezcla de callback y promesa sin control claro del flujo
function obtenerDatos(cb) {
  fetch('/api/datos')
    .then(res => res.json())
    .then(data => cb(null, data))
    .catch(err => cb(err));
}
```

```
//-- Uso consistente de async/await para claridad y control
async function obtenerDatos() {
  const res = await fetch('/api/datos');
  return await res.json();
}
```

- Evitar encadenar promesas dentro de `callbacks`; convertir APIs basadas en `callbacks` a promesas antes de integrarlas.

```
//-- Callback que invoca promesa generando complejidad innecesaria
leerArchivo('config.json', (err, data) => {
  if (err) throw err;
  procesarDatos(data).then(resultado => {
```

```
    guardarResultado(resultado);
  });
});
```

```
//-- Conversión previa a promesa para un flujo uniforme
const { promisify } = require('util');
const leerArchivoAsync = promisify(leerArchivo);
async function ejecutar() {
  const data = await leerArchivoAsync('config.json');
  const resultado = await procesarDatos(data);
  guardarResultado(resultado);
}
```

- Centralizar el manejo de errores en un único mecanismo para evitar omisiones o dobles invocaciones.

```
//-- Manejo de errores disperso entre callback y promesa
function procesar(cb) {
  obtenerToken((err, token) => {
    if (err) return cb(err);
    validarToken(token).then(() => {
      cb(null, 'OK');
    }).catch(e => cb(e));
  });
}
```

```
//-- Manejo de errores unificado con try/catch
async function procesar() {
  try {
    const token = await obtenerTokenAsync();
    await validarToken(token);
    return 'OK';
  } catch (e) {
    throw e;
  }
}
```

2.5.3.2 Estándares metodológicos

- **Airbnb-JS-7**: Functions.
- **MDN Web Docs JS-15**: Promises.

2.5.4 [js.sync.4] - No pasar cadenas de texto a "setTimeout" o "setInterval"

ID	**js.sync.4**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las funciones `setTimeout` y `setInterval` pueden ejecutar código de forma dinámica cuando reciben cadenas de texto como primer parámetro, comportándose de manera similar a `eval`. Esta práctica introduce **vulnerabilidades de seguridad** al permitir la ejecución de código arbitrario y dificulta el análisis estático del programa.

Codificación insegura

Uso de temporizadores para ejecutar una acción diferida en JavaScript. Se muestra cómo programar una operación que imprime información en consola después de un tiempo determinado, permitiendo entender los riesgos de evaluar código dinámicamente cuando la fuente puede ser manipulada.

```
const userInput = "console.log('Mensaje ejecutado');";

function ejecutarConRetraso() {
  //-- Se utiliza una cadena que será evaluada dinámicamente
  setTimeout(userInput, 1000);
}
//-- Punto de entrada del programa
ejecutarConRetraso();
```

Codificación recomendada

Uso de temporizadores para ejecutar una acción diferida en JavaScript evitando la evaluación dinámica de cadenas. Se muestra cómo pasar una función como referencia, reduciendo el riesgo de inyección de código y mejorando la seguridad y mantenibilidad.

```
function tareaSegura() {
  //-- Acción controlada definida como función
  console.log("Mensaje ejecutado");
}
function ejecutarConRetraso() {
  //-- Se pasa una función en lugar de una cadena
  setTimeout(tareaSegura, 1000);
}
//-- Punto de entrada del programa
ejecutarConRetraso();
```

2.5.4.1 Descripción

Las funciones `setTimeout` y `setInterval` pueden recibir como primer parámetro tanto una función como una cadena de texto. Cuando se proporciona una cadena, el motor de ejecución la interpreta como código que debe ser evaluado dinámicamente, comportándose de manera similar a `eval`.

Esta evaluación dinámica de cadenas introduce **vulnerabilidades de seguridad significativas**, especialmente cuando las cadenas contienen datos controlados por el usuario o provienen de fuentes externas no confiables. El código malicioso puede ser inyectado y ejecutado en el contexto de la aplicación, comprometiendo la integridad del sistema.

Además de los riesgos de seguridad, la evaluación de cadenas presenta **desventajas en el rendimiento**. El motor debe parsear y compilar el código en tiempo de ejecución, lo que genera overhead adicional comparado con la ejecución directa de funciones predefinidas.

La evaluación dinámica también **dificulta el análisis estático del código**, impidiendo que las herramientas de desarrollo detecten errores potenciales, realicen optimizaciones o proporcionen autocompletado preciso. Los depuradores y analizadores de código no pueden rastrear eficientemente el flujo de ejecución cuando se evalúan cadenas dinámicamente.

Para mitigar estos riesgos, se debe **utilizar exclusivamente referencias a funciones** como primer parámetro de `setTimeout` y `setInterval`. Las funciones pueden ser definidas previamente, declaradas inline como funciones anónimas, o implementadas como funciones flecha. Los parámetros adicionales necesarios pueden pasarse como argumentos posteriores a estas funciones temporales.

Esta práctica garantiza que el código sea **predecible, seguro y eficiente**, eliminando los vectores de ataque asociados con la evaluación dinámica de cadenas y manteniendo la compatibilidad con las herramientas de desarrollo modernas.

Riesgos asociados

- Ejecución de código arbitrario, lo que podría permitir a un atacante ejecutar código malicioso en el contexto de la aplicación.
- Exposición a ataques Cross-Site Scripting (XSS) si los datos de usuario se insertan sin validación.
- Dificultad para aplicar herramientas de análisis estático y detección de vulnerabilidades.
- Obstáculos para la mantenibilidad y legibilidad del código.
- Incremento de la probabilidad de errores y fallos de seguridad inadvertidos.
- Posibilidad de acceso o manipulación de variables fuera del alcance esperado

(scope pollution).

- Compromiso de la integridad de la aplicación y fuga de información sensible.

Buenas prácticas

- Pasar funciones como referencia a `setTimeout` y `setInterval` en lugar de cadenas de texto para evitar evaluación dinámica.

```
//-- Uso de cadena que implica evaluación dinámica similar a eval
setTimeout("procesarDatos()", 1000);
```

```
//-- Uso de referencia a función evitando evaluación dinámica
setTimeout(procesarDatos, 1000);
```

- Evitar construir código dinámicamente concatenando variables dentro de cadenas usadas en temporizadores.

```
//-- Concatenación de datos en cadena evaluada dinámicamente
var usuario = "admin";
setInterval("auditar('" + usuario + "')", 5000);
```

```
//-- Paso de parámetros mediante función anónima sin evaluación dinámica
var usuario = "admin";
setInterval(function() {
  auditar(usuario);
}, 5000);
```

- Utilizar funciones flecha para mantener claridad y evitar ejecución implícita de código en texto.

```
//-- Ejecución de expresión en texto interpretado dinámicamente
setTimeout("console.log(calcularTotal())", 2000);
```

```
//-- Uso de función flecha como callback seguro
setTimeout(() => {
  console.log(calcularTotal());
}, 2000);
```

- Validar y encapsular la lógica en funciones definidas previamente en lugar de depender de nombres globales evaluados como texto.

```
//-- Invocación mediante nombre global evaluado como texto
setTimeout("inicializarSistema()", 3000);
```

```
//-- Encapsulamiento en función definida y pasada como referencia
function inicializarSistema() {
  configurar();
}
setTimeout(inicializarSistema, 3000);
```

2.5.4.2 Estándares metodológicos

- **JS-WASCP-14.2**: Interpreted Code Integrity.
- **Google-JS-5.11**: Disallowed features.

2.5.5 [js.sync.5] - Controlar explícitamente el ciclo de vida de setInterval y temporizadores

ID	js.sync.5	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los temporizadores e intervalos que no se gestionan correctamente pueden continuar ejecutándose indefinidamente, consumiendo recursos del sistema y provocando **fugas de memoria**. La falta de control sobre estos mecanismos genera comportamientos impredecibles en aplicaciones, especialmente cuando los componentes se destruyen o las páginas cambian de contexto sin limpiar adecuadamente los procesos en segundo plano.

Codificación insegura

Uso de `setInterval` para ejecutar una tarea periódica sin ningún control sobre su ciclo de vida. La intención es mostrar cómo una tarea repetitiva puede quedar ejecutándose indefinidamente, incluso cuando ya no es necesaria, generando consumo innecesario de memoria y CPU. Se pretende observar cómo la falta de cancelación explícita provoca ejecución no controlada.

```
//-- Simula una tarea periódica sin control de ciclo de vida
function iniciarProceso() {
  //-- Se crea un intervalo que nunca se detiene
  setInterval(() => {
    //-- Tarea repetitiva
    console.log('Procesando tarea cada segundo...');
  }, 1000);
}
//-- Se inicia el proceso
iniciarProceso();

//-- El proceso principal continúa sin mecanismo para detener el intervalo
```

```
setTimeout(() => {
  console.log('El flujo principal ha terminado, pero el intervalo sigue activo');
}, 5000);
```

Codificación recomendada

Uso de `setInterval` con control explícito del identificador del temporizador y mecanismos claros de cancelación. La intención es mostrar cómo gestionar correctamente el ciclo de vida del intervalo, asegurando que se detenga cuando ya no es necesario y evitando ejecución indefinida o fugas de recursos.

```
'use strict';

//-- Variable para almacenar el identificador del intervalo
let intervalId = null;

//-- Función que inicia el proceso controlado
function iniciarProceso() {
  //-- Evita crear múltiples intervalos si ya existe uno activo
  if (intervalId !== null) {
    return;
  }
  intervalId = setInterval(() => {
    //-- Tarea repetitiva
    console.log('Procesando tarea cada segundo...');
  }, 1000);
}
//-- Función que detiene el proceso de forma explícita
function detenerProceso() {
  if (intervalId !== null) {
    clearInterval(intervalId);
    intervalId = null;
    console.log('Intervalo detenido correctamente');
  }
}
//-- Se inicia el proceso
iniciarProceso();

//-- Se programa la detención controlada tras 5 segundos
setTimeout(() => {
  detenerProceso();
}, 5000);
```

2.5.5.1 Descripción

Los temporizadores representan recursos del sistema que requieren gestión explícita de su ciclo de vida. Cuando se crean mediante `setInterval`, `setTimeout` u otros

mecanismos similares, estos permanecen activos hasta que se cancelen de forma explícita o se cierre el contexto de ejecución.

La gestión inadecuada de estos recursos genera **fugas de memoria** y **ejecución no controlada de código**. Los temporizadores no cancelados continúan ejecutándose en segundo plano, consumiendo recursos del sistema y potencialmente accediendo a objetos que ya no deberían estar disponibles.

El problema se agrava en aplicaciones de larga duración, donde la acumulación de temporizadores activos puede degradar significativamente el rendimiento. En entornos de navegador, esto afecta la experiencia del usuario mediante ralentización de la interfaz y consumo excesivo de batería en dispositivos móviles.

Los riesgos se intensifican cuando los temporizadores mantienen referencias a objetos del DOM o estructuras de datos que han sido eliminadas conceptualmente de la aplicación. Esta situación crea **referencias colgantes** que impiden la recolección de basura y pueden provocar comportamientos impredecibles.

La mitigación requiere implementar un control explícito sobre la creación y destrucción de temporizadores. Esto implica almacenar las referencias devueltas por las funciones de temporización y utilizar `clearInterval` o `clearTimeout` cuando el temporizador ya no sea necesario.

En arquitecturas basadas en componentes, la gestión debe integrarse con el ciclo de vida del componente, cancelando todos los temporizadores activos durante la fase de limpieza. Los patrones de diseño que encapsulan la lógica de temporización facilitan esta gestión centralizada.

La implementación de mecanismos de seguimiento permite identificar temporizadores huérfanos durante el desarrollo y las pruebas. Estas herramientas ayudan a detectar casos donde la cancelación no se ejecuta correctamente debido a flujos de control complejos o manejo de excepciones.

Riesgos asociados

- Fugas de memoria debido a temporizadores no cancelados.
- Ejecución continua e incontrolada de funciones, provocando uso excesivo de CPU.
- Comportamiento inesperado de la aplicación por ejecución de lógica obsoleta o innecesaria.
- Incremento en el consumo de recursos del navegador o servidor.
- Posibilidad de ataques de denegación de servicio (DoS) por saturación de recursos.
- Dificultad para depurar y mantener el código a largo plazo.

- Pérdida de referencias a objetos, impidiendo su recolección por el recolector de basura.
- Filtración de información sensible si las funciones vinculadas al temporizador acceden a datos confidenciales más allá de su ciclo de vida previsto.

Buenas prácticas

- Guardar el identificador devuelto por `setInterval` y limpiar explícitamente el temporizador cuando ya no sea necesario.

```
//-- Intervalo creado sin almacenar referencia ni liberarlo
setInterval(() => {
  procesarDatos();
}, 1000);
```

```
//-- Intervalo almacenado y liberado explícitamente
const intervalId = setInterval(() => {
  procesarDatos();
}, 1000);
if (condicionDeParada) {
  clearInterval(intervalId);
}
```

- Evitar crear múltiples temporizadores repetidos sin control previo del estado.

```
//-- Creación repetida de intervalos sin validar si ya existe uno activo
function iniciar() {
  setInterval(() => {
    sincronizar();
  }, 5000);
}
```

```
//-- Verificación previa y reutilización del identificador existente
let intervalId;
function iniciar() {
  if (!intervalId) {
    intervalId = setInterval(() => {
      sincronizar();
    }, 5000);
  }
}
```

- Cancelar temporizadores al destruir componentes o finalizar recursos para prevenir fugas de memoria.

```
//-- Intervalo asociado a objeto sin mecanismo de limpieza
```

```
class Monitor {
  iniciar() {
    this.intervalId = setInterval(() => {
      this.verificar();
    }, 2000);
  }
}
```

```
//-- Implementación de método explícito para liberar el recurso
class Monitor {
  iniciar() {
    this.intervalId = setInterval(() => {
      this.verificar();
    }, 2000);
  }
  detener() {
    if (this.intervalId) {
      clearInterval(this.intervalId);
      this.intervalId = null;
    }
  }
}
```

- Preferir `setTimeout` recursivo controlado cuando se requiera validar condiciones antes de reprogramar la ejecución.

```
//-- Intervalo continuo sin validar estado antes de cada ejecución
const intervalId = setInterval(() => {
  procesarTarea();
}, 3000);
```

```
//-- Reprogramación controlada según condición dinámica
function ejecutar() {
  if (!condicionActiva) {
    return;
  }
  procesarTarea();
  setTimeout(ejecutar, 3000);
}
ejecutar();
```

2.5.5.2 Estándares metodológicos

- **JS-WASCP-14.4**: Concurrency.
- **MDN Web Docs JS-15**: Promises.

3. Anexo: Metodología de cálculo del score CWSS

Para la priorización de reglas de codificación segura se ha adoptado un perfil simplificado de CWSS 1.0. Este perfil se utiliza únicamente con fines de comparación interna entre reglas y no pretende modelar un hallazgo desplegado en un entorno concreto.

CWSS define tres grupos de métricas cuyo score final es el producto de los tres subscores:

- Base Finding
- Attack Surface
- Environmental

Las métricas del grupo Attack Surface se fijan a valores de peso 1.0. Con esta parametrización, el subscore de Attack Surface es 1.0 y no pondera sobre Base Finding.

- Required Privilege=None
- Required Privilege Layer=Application
- Access Vector=Internet
- Authentication Strength=None
- Level of Interaction=Automated
- Deployment Scope=N/A.

Las métricas del grupo Environmental se fijan a Not Applicable (N/A), ya que describen factores dependientes del entorno operativo, del negocio o de controles externos, información que no forma parte de una guía genérica de implementación. En CWSS, el valor N/A equivale a peso 1.0 cuando la métrica se ignora intencionalmente, por lo que este grupo no modifica la puntuación.

Dentro de Base Finding, la métrica variable es Technical Impact (TI), al ser la única que puede evaluarse de forma estable y portable para reglas de implementación genéricas.

Su valor varía entre 0 y 100, y el literal que se muestra depende de este valor según la siguiente tabla:

Muy alto	>= 80
Alto	>= 60
Medio	>= 40
Bajo	>= 20
Muy bajo	> 0
Ninguno	0

Valores del Impacto Técnico a partir del CWSS Score

El resto de las métricas se fijan a los siguientes valores como convención para su perfil:

- Acquired Privilege=Unknown, porque, en una regla genérica, no siempre puede inferirse de forma consistente qué privilegios concretos se obtendrían en explotación.
- Acquired Privilege Layer=Application, por ser la capa natural de evaluación de la guía.
- Internal Control Effectiveness=None, para no introducir compensaciones propias de un producto concreto.
- Finding Confidence=Confirmed, porque el documento puntúa incumplimientos normativos de reglas, no hallazgos con incertidumbre analítica.

Con estas fijaciones, el score final siempre coincidirá con el subscore del Base Finding.

CWSS Score	**86.00**								
CWSS Vector	CWSS:1.0.1/TI:H,0.90/AP:UK,0.50/AL:A,1.00/IC:N,1.00/FC:T,1.00/RP:N,1.00/RL:A,1.00/AV:I,1.00/AS:N,1.00/IN:A,1.00/SC:A,1.00/BI:NA,1.00/DI:NA,1.00/EX:NA,1.00/EC:NA,1.00/P:NA,1.00								
Hallazgo Base	**86.000**								
Impacto Técnico	Crítico	**Alto**	Medio	Bajo	Ninguno	-	Por Defecto	Desconocido	N/A
Privilegios Adquiridos	Administrador	Usuario Parcialmente Privilegiado	Usuario Regular	Limitado / Invitado	Ninguno	-	Por Defecto	**Desconocido**	N/A
Capa de Privilegios Adquirida	**Aplicación**	Sistema	Red	Infraestructura Empresarial	-	-	Por Defecto	Desconocido	N/A
Eficacia de Controles Internos	**Ninguno**	Limitado	Moderado	Indirecto (Defensa en Profundidad)	Mejor Disponible	Completo	Por Defecto	Desconocido	N/A
Confianza del Hallazgo	**Confirmado**	Confirmado Localmente	Falso Positivo	-	-	-	Por Defecto	Desconocido	N/A
Superficie de Ataque	**1.000**								
Privilegios Requeridos	**Ninguno**	Limitado / Invitado	Usuario Regular	Usuario Parcialmente Privilegiado	Administrador	-	Por Defecto	Desconocido	N/A
Capa de Privilegios Requerida	**Aplicación**	Sistema	Red	Infraestructura Empresarial	-	-	Por Defecto	Desconocido	N/A
Vector de Acceso	**Internet**	Intranet	Red Privada	Red Adyacente	Local	Físico	Por Defecto	Desconocido	N/A
Fortaleza de Autenticación	**Ninguna**	Débil	Moderada	Fuerte	-	-	Por Defecto	Desconocido	N/A
Nivel de Interacción	**Automatizado**	Típica / Limitada	Moderada	Oportunista	Alta	Sin Interacción	Por Defecto	Desconocido	N/A
Alcance de Despliegue	**Todos**	Moderado	Raro	Potencialmente Alcanzable	-	-	Por Defecto	Desconocido	N/A
Entorno	**1.000**								
Impacto en el Negocio	Crítico	Alto	Medio	Bajo	Ninguno	-	Por Defecto	Desconocido	**N/A**
Probabilidad de Descubrimiento	Alta	Media	Baja	-	-	-	Por Defecto	Desconocido	**N/A**
Probabilidad de Explotación	Alta	Media	Baja	Ninguna	-	-	Por Defecto	Desconocido	**N/A**
Eficacia de Controles Externos	Ninguno	Limitado	Moderado	Indirecto (Defensa en Profundidad)	Mejor Disponible	Completo	Por Defecto	Desconocido	**N/A**
Prevalencia	Generalizada	Alta	Común	Limitada	-	-	Por Defecto	Desconocido	**N/A**

CWE and CWSS are trademarks of The MITRE Corporation.

Detalle de las métricas del CWSS con sus valores prefijados, salvo Impacto Técnico.

PUBLICACIONES DE DEFENSA

GOBIERNO DE ESPAÑA
MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA
SUBDIRECCIÓN GENERAL DE PUBLICACIONES Y PATRIMONIO CULTURAL