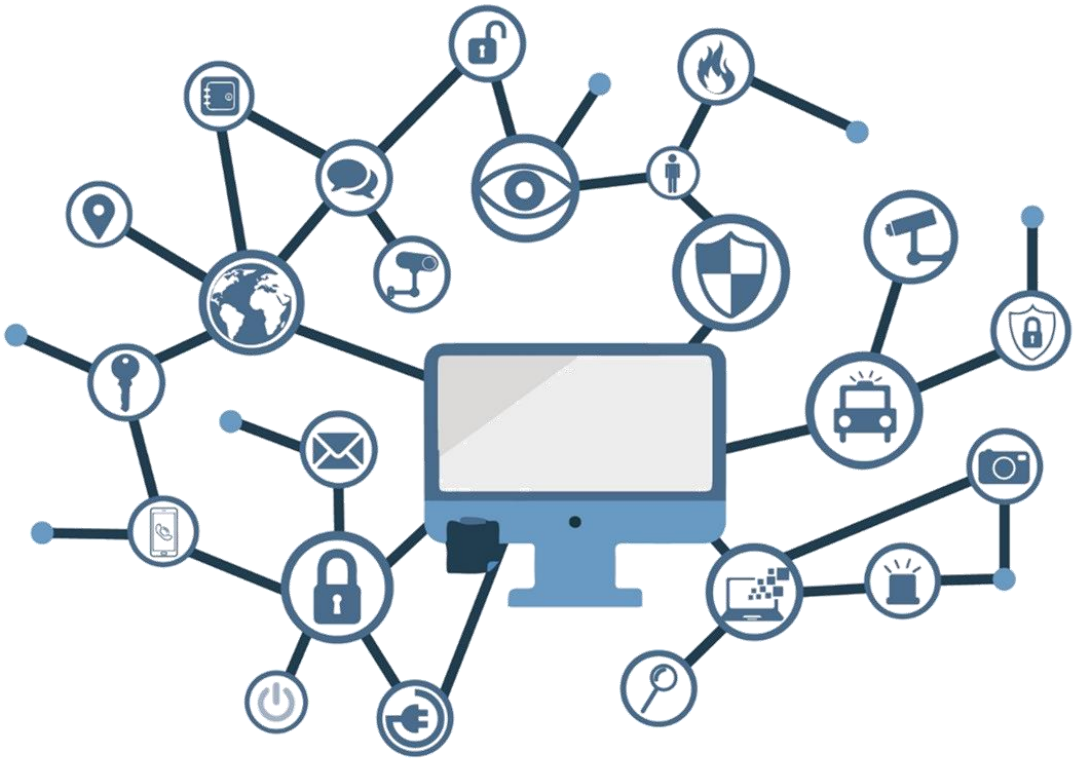
Guía de Desarrollo Seguro
CCN-STIC 481 – Anexo A

Codificación Segura para C



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 MISRA C 2004 Guidelines for the use of the C language in critical systems 2
1.3.2 SEI/CERT C Coding Standard 2
1.4 ESTRUCTURA DE LA GUÍA 3

2. REGLAS DE CODIFICACIÓN SEGURA 5

2.1 CADENAS DE CARACTERES 5
2.1.1 [c.str.1] - No modificar literales de cadena 5
2.1.2 [c.str.2] - Garantizar espacio suficiente para cadenas y su terminador nulo 8
2.1.3 [c.str.3] - No pasar secuencias no terminadas en nulo a funciones que esperan cadenas . 12
2.1.4 [c.str.4] - Convertir caracteres a unsigned char antes de ampliarlos a enteros mayores ... 16
2.1.5 [c.str.5] - Garantizar que los argumentos a funciones de caracteres sean representables como unsigned char 19
2.1.6 [c.str.6] - No confundir cadenas y funciones de caracteres estrechos y anchos 23
2.2 CONVERSIONES 27
2.2.1 [c.conv.1] - Evitar wraparound en operaciones con enteros sin signo 27
2.2.2 [c.conv.2] - Evitar pérdida o mala interpretación de datos en conversiones entre enteros 31
2.2.3 [c.conv.3] - Evitar desbordamientos en operaciones con enteros con signo 35
2.2.4 [c.conv.4] - Evitar división por cero en divisiones enteras 40
2.2.5 [c.conv.5] - No desplazar operandos un número inválido de bits 44
2.2.6 [c.conv.6] - No convertir punteros a tipos con alineación más estricta 48
2.3 INICIALIZACIÓN 52
2.3.1 [c.ini.1] - Inicializar todas las variables automáticas antes de su uso 52
2.3.2 [c.ini.2] - Declarar objetos con duraciones de almacenamiento apropiadas 55
2.3.3 [c.ini.3] - Usar la sintaxis correcta para miembros de arrays flexibles 60
2.3.4 [c.ini.4] - No filtrar información al pasar estructuras a través de límites de confianza 64
2.3.5 [c.ini.5] - No declarar variables dentro de un switch antes de la primera etiqueta case 68
2.3.6 [c.ini.6] - No acceder a objetos mediante punteros de tipo incompatible 73
2.3.7 [c.ini.7] - No acceder a objetos volatile a través de referencias no volatile 77
2.3.8 [c.ini.8] - No modificar objetos con duración temporal 81
2.4 EXPRESIONES 85
2.4.1 [c.exp.1] - Garantizar que el valor de una expresión sea independiente del orden de evaluación 85
2.4.2 [c.exp.2] - No depender del orden de evaluación para efectos laterales 89
2.5 ESTRUCTURAS DE CONTROL Y FLUJO 93
2.5.1 [c.flow.1] - Asegurar que las construcciones "if/else if" deben terminar con un "else" final 93
2.5.2 [c.flow.2] - Asegurar que las sentencias switch deben tener una estructura completa y controlada 97
2.5.3 [c.flow.3] - Evitar utilizar "goto" o asegurar que solo salta a etiquetas dentro del bloque envolvente 102
2.5.4 [c.flow.4] - Evitar el uso de llamadas recursivas directas o indirectas 105
2.6 FUNCIONES 110
2.6.1 [c.func.1] - Llamar a funciones con el número y tipo correctos de argumentos 110
2.6.2 [c.func.2] - No llamar "va_arg" con un tipo incorrecto 114
2.6.3 [c.func.3] - No llamar a "system" 118
2.6.4 [c.func.4] - No pasar datos inválidos a asctime 122

2.7 SEÑALES 126
2.7.1 [c.sig.1] - Llamar solo a funciones asincrónicamente seguras dentro de manejadores de señales 126
2.7.2 [c.sig.2] - No acceder a objetos compartidos dentro de manejadores de señales 131
2.7.3 [c.sig.3] - No retornar desde manejadores de señales de excepción computacional 135
2.7.4 [c.sig.4] - Asegurar que un manejador de señales sea una función simple 140
2.8 PUNTEROS 144
2.8.1 [c.ptr.1] - Evitar aritmética de punteros fuera del mismo array 144
2.8.2 [c.ptr.2] - No usar punteros a objetos que hayan dejado de existir 148
2.8.3 [c.ptr.3] - No acceder a memoria fuera de los límites de un objeto 153
2.8.4 [c.ptr.4] - No almacenar punteros devueltos por funciones que los invalidan 157
2.8.5 [c.ptr.5] - No desreferenciar punteros nulos 161
2.8.6 [c.ptr.6] - No provocar comportamiento indefinido al usar punteros restrict 165
2.9 GESTIÓN DE MEMORIA 169
2.9.1 [c.mem.1] - No acceder a memoria ya liberada 169
2.9.2 [c.mem.2] - No acceder a memoria no inicializada 173
2.9.3 [c.mem.3] - Liberar la memoria asignada dinámicamente cuando deje de ser necesaria 176
2.9.4 [c.mem.4] - No liberar memoria no asignada dinámicamente 181
2.9.5 [c.mem.5] - Garantizar que se asigne memoria suficiente para un objeto 185
2.10 VALIDACIONES 189
2.10.1 [c.val.1] - No incluir entrada del usuario en cadenas de formato 189
2.10.2 [c.val.2] - Distinguir correctamente entre caracteres leídos y EOF o WEOF 193
2.10.3 [c.val.3] - Evitar condiciones de carrera TOCTOU al acceder a archivos 197
2.10.4 [c.val.4] - No acceder a un archivo cerrado 201
2.10.5 [c.val.5] - Usar únicamente cadenas de formato válidas 205
2.10.6 [c.val.6] - Considerar siempre la posibilidad de entradas malformadas al leer datos . 209
2.10.7 [c.val.7] - Garantizar que los tamaños de arrays de longitud variable estén dentro de rangos válidos 214
2.11 GESTIÓN DE ERRORES 219
2.11.1 [c.err.1] - Detectar y manejar errores de bibliotecas estándar 219
2.11.2 [c.err.2] - Detectar errores al convertir cadenas a números 223
2.12 CONCURRENCIA 227
2.12.1 [c.sync.1] - No destruir un mutex mientras esté bloqueado 227
2.12.2 [c.sync.2] - Prevenir condiciones de carrera de datos en código multihilo 231
2.12.3 [c.sync.3] - Evitar interbloqueos bloqueando mutex en un orden predefinido 236
2.12.4 [c.sync.4] - No volver a unir o desvincular un hilo previamente unido o desvinculado 242
2.13 POSIX 246
2.13.1 [c.posix.1] - No llamar a putenv con punteros a variables automáticas 246
2.13.2 [c.posix.2] - Evitar condiciones de carrera al comprobar enlaces simbólicos 250
2.13.3 [c.posix.3] - Evitar condiciones de carrera al usar fork y descriptores de archivo 254
2.13.4 [c.posix.4] - No usar señales para terminar hilos 259
2.13.5 [c.posix.5] - No usar hilos cancelables asincrónicamente 264
2.14 NÚMEROS ALEATORIOS 269
2.14.1 [c.rand.1] - No usar "rand" para generar números pseudoaleatorios 269
2.14.2 [c.rand.2] - Inicializar correctamente el generador de números aleatorios 274

3. ANEXO: METODOLOGÍA DE CÁLCULO DEL SCORE CWSS 278

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 MISRA C 2004 Guidelines for the use of the C language in critical systems

https://es.parasoft.com/solutions/misra/

Es un conjunto de reglas de codificación para el lenguaje C publicado por el Motor Industry Software Reliability Association (MISRA). Su objetivo es reducir los comportamientos indefinidos, no especificados y dependientes de la implementación que el estándar C permite, y que en sistemas críticos pueden derivar en fallos difíciles de detectar y reproducir.

Las reglas cubren áreas como el uso de tipos, conversiones implícitas, aritmética de punteros, control de flujo, uso de funciones de la biblioteca estándar y comportamiento en tiempo de compilación.

MISRA C 2004 es ampliamente adoptado en sectores como automoción, aeronáutica, ferroviario y dispositivos médicos, donde la fiabilidad del software es un requisito normativo. Aunque en 2012 se publicó una revisión posterior (MISRA C:2012), la edición de 2004 sigue presente en proyectos con ciclos de vida largos o requisitos contractuales que la referencian explícitamente.

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

- **MISRA C-12.2**: The value of an expression shall be the same under any order of evaluation that the standard permits.
- **MISRA C-17.4**: Array indexing shall be the only allowed form of pointer arithmetic.
- **MISRA C-20.9**: The input/output library <stdio.h> shall not be used in production code.

1.3.2 SEI/CERT C Coding Standard

https://wiki.sei.cmu.edu/confluence/spaces/c/pages/87152044/SEI+CERT+C+Coding+Standard

Es un conjunto de reglas y recomendaciones para el desarrollo seguro en C, publicado por el Software Engineering Institute (SEI) de la Universidad Carnegie Mellon a través de su programa CERT. Su objetivo principal es eliminar patrones de codificación que introducen vulnerabilidades de seguridad explotables, como desbordamientos de búfer, condiciones de carrera, uso de memoria después de liberarla o inyección de datos.

Las áreas que cubre incluyen gestión de memoria dinámica, operaciones con cadenas, aritmética de enteros (con especial atención al desbordamiento y la conversión entre tipos con y sin signo), entrada/salida, señales, concurrencia y uso de la biblioteca estándar. También trata el comportamiento indefinido de C de forma explícita, catalogando los casos más peligrosos.

CERT C se orienta hacia la **seguridad frente a ataques externos**, por lo que es especialmente relevante en software de infraestructura, sistemas operativos, aplicaciones de red y cualquier entorno donde el código procese datos no confiables.

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

- **CERT/C-FIO30**: Exclude user input from format strings.
- **CERT/C-ARR32**: Ensure size arguments for variable length arrays are in a valid range.
- **CERT/C-CON32**: Prevent data races when accessing bit-fields from multiple threads.

1.4 Estructura de la guía

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

- Cadenas de caracteres.
- Conversiones.
- Inicialización.
- Expresiones.
- Estructuras de control y flujo.
- Funciones.
- Señales.
- Punteros.
- Gestión de memoria.
- Validaciones.
- Gestión de errores.
- Concurrencia.
- POSIX.
- Números aleatorios.

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 Cadenas de caracteres

2.1.1 [c.str.1] - No modificar literales de cadena

ID	c.str.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los literales de cadena se almacenan en memoria de solo lectura y cualquier intento de modificación podría provocar fallos de segmentación, corrupción de datos o vulnerabilidades de seguridad, ya que el compilador no garantiza la ubicación ni protección de estos literales en memoria.

Codificación insegura

Se muestra cómo trabajar con cadenas en C cuando se almacenan en memoria de solo lectura. El objetivo es observar qué ocurre cuando se intenta modificar directamente un literal de cadena y entender el comportamiento en tiempo de ejecución.

```
#include <stdio.h>
#include <string.h>

int main() {
  //-- Se declara un puntero a un literal de cadena (almacenado en memoria de solo
lectura)
  char *mensaje = "Hola Mundo";

  //-- Se muestra el contenido original
  printf("Mensaje original: %s\n", mensaje);

  //-- Se intenta modificar el primer caracter del literal
  //-- Este comportamiento es indefinido y puede provocar un fallo de segmentación
  mensaje[0] = 'h';

  //-- Se muestra el contenido despues de la modificacion
  printf("Mensaje modificado: %s\n", mensaje);

  return 0;
}
```

Codificación recomendada

Se presenta el uso correcto de cadenas cuando se necesita modificar su contenido. El objetivo es almacenar la cadena en memoria escribible y realizar cambios de forma segura y definida por el estándar del lenguaje.

```
#include <stdio.h>
#include <string.h>

int main() {
  //-- Se declara un array de caracteres inicializado con un literal
  //-- En este caso el contenido se copia en memoria escribible
  char mensaje[] = "Hola Mundo";

  //-- Se muestra el contenido original
  printf("Mensaje original: %s\n", mensaje);

  //-- Se modifica el primer caracter de forma segura
  mensaje[0] = 'h';

  //-- Se muestra el contenido despues de la modificacion
  printf("Mensaje modificado: %s\n", mensaje);

  return 0;
}
```

2.1.1.1 Descripción

Los literales de cadena representan secuencias de caracteres almacenadas en memoria de solo lectura durante la ejecución del programa. El estándar define estos literales como objetos con duración de almacenamiento estática, ubicados típicamente en segmentos de memoria protegidos contra escritura.

La modificación de literales de cadena constituye **comportamiento indefinido** según el estándar. Aunque algunos compiladores y sistemas pueden permitir estas operaciones sin generar errores inmediatos, el comportamiento resultante no está garantizado y puede variar entre diferentes implementaciones, versiones del compilador o configuraciones del sistema.

Los riesgos asociados incluyen **fallos de segmentación** cuando el sistema operativo detecta intentos de escritura en memoria protegida, **corrupción silenciosa de datos** en implementaciones que no aplican protecciones estrictas, y **comportamiento impredecible** que puede manifestarse de forma intermitente o dependiente del entorno de ejecución.

La distinción entre punteros a literales de cadena y punteros a arrays de caracteres modificables resulta fundamental para prevenir estos problemas. Los literales de cadena

se declaran implícitamente como `const char[]`, mientras que los arrays de caracteres inicializados con contenido de cadena permiten modificaciones posteriores.

Las estrategias de mitigación se centran en el **uso explícito del calificador** `const` para punteros que referencian literales de cadena, lo que permite al compilador detectar intentos de modificación durante la compilación. Cuando se requiere manipulación de contenido de cadena, se debe optar por la **copia del literal a un array modificable** antes de realizar las operaciones de escritura.

Los compiladores modernos ofrecen opciones de diagnóstico que pueden detectar asignaciones potencialmente problemáticas de literales de cadena a punteros no constantes, facilitando la identificación temprana de violaciones de esta regla durante el desarrollo.

Riesgos asociados

- Comportamiento indefinido del programa al intentar modificar memoria de solo lectura.
- Fallos de segmentación (segmentation fault) y bloqueos inesperados.
- Vulnerabilidad a ataques de denegación de servicio (DoS) por caída del programa.
- Dificultad en la depuración y mantenimiento debido a errores no reproducibles.
- Dependencia de comportamientos no portables que varían entre plataformas y compiladores.
- Corrupción de datos compartidos si múltiples partes del programa referencian el mismo literal.

Buenas prácticas

- Declarar los literales de cadena como `const char *` para evitar su modificación accidental.

```
//-- Se declara un puntero a char que permite modificar el literal
char *mensaje = "Hola";
mensaje[0] = 'h';
```

```
//-- Se declara el puntero como const para impedir modificaciones
const char *mensaje = "Hola";
/* mensaje[0] = 'h'; */
```

- Copiar el literal a un array modificable cuando sea necesario alterar su contenido.

```
//-- Se intenta modificar directamente el literal almacenado en memoria de solo
lectura
```

```
char *ruta = "/tmp/data";
ruta[0] = 'T';
```

```
//-- Se copia el literal en un array local que permite modificaciones
char ruta[] = "/tmp/data";
ruta[0] = 'T';
```

- Evitar pasar literales a funciones que reciben parámetros `char *` y podrían modificarlos.

```
//-- La función recibe un puntero modificable y altera el contenido
void convertirMayusculas(char *texto) {
  texto[0] = 'X';
}
convertirMayusculas("ejemplo");
```

```
//-- La función define el parámetro como const para garantizar inmutabilidad
void convertirMayusculas(const char *texto) {
  /* solo lectura */
}
convertirMayusculas("ejemplo");
```

2.1.1.2 Estándares metodológicos

- **CERT/C-STR30**: Do not attempt to modify string literals.

2.1.2 [c.str.2] - Garantizar espacio suficiente para cadenas y su terminador nulo

ID	c.str.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los desbordamientos de buffer representan una de las vulnerabilidades más frecuentes en aplicaciones. Cuando trabajamos con cadenas de caracteres, debemos asegurar que los buffers de destino tengan **espacio suficiente** para almacenar tanto el contenido de la cadena como su terminador nulo. La omisión de esta verificación puede provocar **corrupción de memoria**, comportamiento impredecible del programa y potenciales brechas de seguridad explotables por atacantes.

Codificación insegura

Uso de buffers para almacenar cadenas en memoria, mostrando cómo un tamaño insuficiente puede provocar desbordamientos y corrupción de memoria al no considerar

el terminador nulo. Se observa el riesgo que supone definir incorrectamente el tamaño del array y usar directamente funciones de copia, así como su impacto en ejecución.

```
#include <stdio.h>
#include <string.h>

//-- Programa que copia una cadena en un buffer sin espacio suficiente
int main() {
  //-- Cadena de 8 caracteres más el terminador nulo '\0'
  const char *mensaje = "SEGURIDAD";

  //-- Buffer insuficiente: solo 8 bytes, no hay espacio para '\0'
  char buffer[8];

  //-- Copia insegura que puede provocar desbordamiento
  strcpy(buffer, mensaje);

  //-- Impresión del contenido (comportamiento indefinido)
  printf("Contenido del buffer: %s\n", buffer);

  return 0;
}
```

Codificación recomendada

Reserva adecuada de memoria para una cadena, considerando explícitamente el espacio para el terminador nulo y utilizando una función de copia que limita la cantidad de bytes escritos. Se muestra cómo calcular el tamaño necesario y evitar desbordamientos de memoria.

```
#include <stdio.h>
#include <string.h>

//-- Programa que garantiza espacio suficiente para la cadena y su terminador nulo
int main() {
  //-- Cadena de 8 caracteres
  const char *mensaje = "SEGURIDAD";

  //-- Cálculo del tamaño necesario incluyendo el terminador nulo
  size_t longitud = strlen(mensaje);
  char buffer[9];

  //-- Copia segura limitando el número de caracteres
  snprintf(buffer, sizeof(buffer), "%s", mensaje);

  //-- Impresión segura del contenido
  printf("Contenido del buffer: %s\n", buffer);
```

```
  return 0;
}
```

2.1.2.1 Descripción

Las operaciones con cadenas de caracteres requieren que los buffers de destino tengan capacidad suficiente para almacenar tanto los caracteres de datos como el terminador nulo que marca el final de la cadena. Esta regla establece que debemos verificar y garantizar que existe espacio adecuado antes de realizar cualquier operación de copia, concatenación o manipulación de cadenas.

El problema surge porque las cadenas utilizan un terminador nulo `\0` para indicar su final, pero este carácter adicional no siempre se considera al calcular el espacio necesario. Cuando el buffer de destino no tiene capacidad suficiente, las operaciones de cadenas pueden escribir más allá de los límites asignados, causando corrupción de memoria adyacente.

Las funciones estándar como `strcpy`, `strcat`, `sprintf` y similares no realizan verificaciones automáticas de límites, por lo que la responsabilidad recae completamente en el programador. Esta característica las convierte en fuentes frecuentes de vulnerabilidades cuando no se utilizan correctamente.

Los riesgos asociados incluyen **desbordamientos de buffer**, que pueden permitir la ejecución de código arbitrario, corrupción de datos en memoria, y fallos del programa. Estos problemas son especialmente críticos en aplicaciones que procesan datos de entrada no confiables.

La mitigación efectiva implica calcular correctamente el tamaño requerido incluyendo el terminador nulo, utilizar funciones que permiten especificar límites como `strncpy` o `snprintf`, y validar siempre que el buffer de destino tenga capacidad suficiente antes de realizar la operación. También resulta útil inicializar los buffers y verificar los valores de retorno de las funciones que indican si la operación se completó exitosamente.

Riesgos asociados

- Desbordamiento de buffer (buffer overflow).
- Ejecución de código arbitrario.
- Vulnerabilidad a ataques de ejecución remota de código (RCE).
- Corrupción de memoria y datos.
- Divulgación de información sensible.
- Fallos inesperados y comportamiento indeterminado de la aplicación.
- Error en la manipulación de cadenas (truncada o sin terminador).

- Condiciones de carrera y problemas de sincronización.
- Problemas de integridad de los datos procesados.
- Incremento de la superficie de ataque del software.

Buenas prácticas

- Dimensionar los buffers considerando el tamaño máximo de datos más el terminador nulo.

```
//-- Buffer sin espacio para el terminador nulo
char nombre[5];
strcpy(nombre, "Carlos");
```

```
//-- Buffer dimensionado con espacio adicional para '\0'
char nombre[7];
strcpy(nombre, "Carlos");
```

- Calcular el tamaño del buffer usando `strlen()` más uno para el terminador.

```
//-- Reserva dinámica sin considerar el terminador nulo
char *msg = malloc(strlen(texto));
strcpy(msg, texto);
```

```
//-- Reserva dinámica incluyendo espacio para '\0'
char *msg = malloc(strlen(texto) + 1);
strcpy(msg, texto);
```

- Usar funciones seguras indicando explícitamente el tamaño total del buffer.

```
//-- Copia sin validar límites del buffer destino
char destino[10];
strcpy(destino, origen);
```

```
//-- Copia limitada al tamaño del buffer incluyendo el terminador
char destino[10];
snprintf(destino, sizeof(destino), "%s", origen);
```

- Validar que el tamaño definido como constante contemple el terminador nulo.

```
//-- Constante definida sin considerar el terminador
#define TAM 8
char clave[TAM];
strcpy(clave, "12345678");
```

```
//-- Constante definida considerando espacio para '\0'
#define TAM 9
char clave[TAM];
strcpy(clave, "12345678");
```

- Al truncar manualmente con `strncpy()`, asegurar la inserción explícita del terminador.

```
//-- Copia que puede dejar la cadena sin terminador nulo
char buffer[6];
strncpy(buffer, entrada, sizeof(buffer));
```

```
//-- Copia con terminación explícita del buffer
char buffer[6];
strncpy(buffer, entrada, sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0';
```

2.1.2.2 Estándares metodológicos

- **CERT/C-STR31**: Guarantee that storage for strings has sufficient space for character data and the null terminator.

2.1.3 [c.str.3] - No pasar secuencias no terminadas en nulo a funciones que esperan cadenas

ID	c.str.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de manipulación de cadenas asumen que los datos recibidos están **terminados en nulo**, pero los arrays de caracteres no siempre cumplen esta condición. Pasar secuencias sin el terminador `\0` a funciones como `strlen`, `strcpy` o `printf` provoca **lecturas fuera de límites** que pueden corromper memoria, causar fallos del programa o exponer información sensible.

Codificación insegura

Se muestra cómo el uso de un array de caracteres que no está terminado en nulo puede generar comportamiento indefinido cuando se pasa a funciones que esperan una cadena C válida. El objetivo es observar cómo una omisión en la terminación puede provocar lecturas fuera de límites y resultados impredecibles.

```
#include <stdio.h>
#include <string.h>
```

```
//-- Demostración de paso de secuencia no terminada en nulo a función de cadena
int main() {
  //-- Array de 4 caracteres sin espacio para el terminador nulo
  char buffer[4] = {'t', 'e', 's', 't'};

  //-- printf espera una cadena terminada en '\0'
  //-- Comportamiento indefinido: puede leer memoria fuera del array
  printf("Contenido: %s\n", buffer);

  return 0;
}
```

Codificación recomendada

Se presenta una forma segura de manejar arrays de caracteres asegurando la terminación en nulo antes de pasarlos a funciones que operan con cadenas C. Se busca garantizar que las funciones estándar no lean más allá de los límites definidos del array.

```
#include <stdio.h>
#include <string.h>

//-- Demostración de cadena correctamente terminada en nulo
int main() {
  //-- Array con espacio adicional para el terminador nulo
  char buffer[5];

  //-- Copia segura de datos asegurando el terminador
  memcpy(buffer, "test", 4);
  buffer[4] = '\0';

  //-- Ahora es seguro pasar la cadena a funciones estándar
  printf("Contenido: %s\n", buffer);

  return 0;
}
```

2.1.3.1 Descripción

Cuando se pasan secuencias de caracteres que no terminan en nulo (\0), las funciones continúan procesando memoria más allá del final previsto de la cadena hasta encontrar un terminador nulo o hasta que se produce un fallo de acceso a memoria.

Este comportamiento genera **lecturas fuera de límites** que pueden exponer información sensible de la memoria adyacente o causar fallos de segmentación. Las funciones afectadas incluyen las de la familia `str*` como `strlen`, `strcpy`, `strcmp` y `printf` con especificadores de formato de cadena.

El problema surge frecuentemente al trabajar con **arrays de caracteres inicializados parcialmente**, datos leídos desde archivos o redes sin validación adecuada, o al manipular subcadenas extraídas de cadenas más grandes sin garantizar la terminación nula.

La mitigación requiere **verificar explícitamente la presencia del terminador nulo** antes de pasar datos a funciones de cadena o utilizar alternativas que acepten límites de longitud como las funciones de la familia `strn*`. También resulta efectivo inicializar completamente los arrays de caracteres y validar los datos de entrada antes de su procesamiento.

Esta regla se alinea con las recomendaciones de **SEI CERT STR32-C**, que establece que las cadenas deben estar correctamente terminadas en nulo antes de ser utilizadas como argumentos en funciones que esperan este formato.

Riesgos asociados

- Desbordamiento de buffer debido a la lectura más allá del array al buscar el byte nulo.
- Divulgación de información sensible por acceder a memoria no inicializada.
- Condiciones de denegación de servicio (DoS) al provocar fallos que terminan el programa.
- Comportamiento indefinido, provocando fallos difíciles de reproducir o explotar.
- Vulnerabilidades de seguridad como ejecución de código arbitrario si la memoria leída es controlada por un atacante.
- Corrupción de memoria, haciendo que otras partes del programa se comporten de manera incorrecta.
- Errores de validación y comparación de cadenas al procesar datos inconsistentes o incompletos.

Buenas prácticas

- Asegurar la terminación en nulo al copiar datos a un buffer de caracteres.

```
//-- Copia sin garantizar terminador nulo
char src[5] = {'H','o','l','a','!'};
char dest[5];
memcpy(dest, src, sizeof(src));
printf("%s", dest);
```

```
//-- Copia asegurando terminador nulo explícito
char src2[5] = {'H','o','l','a','!'};
char dest2[6];
memcpy(dest2, src2, sizeof(src2));
```

```
dest2[5] = '\0';
printf("%s", dest2);
```

- Reservar espacio adicional para el carácter nulo al trabajar con buffers dinámicos.

```
//-- Reserva sin espacio para terminador nulo
size_t len = strlen(input);
char *buffer = malloc(len);
memcpy(buffer, input, len);
process_string(buffer);
```

```
//-- Reserva considerando espacio para terminador nulo
size_t len2 = strlen(input);
char *buffer2 = malloc(len2 + 1);
memcpy(buffer2, input, len2);
buffer2[len2] = '\0';
process_string(buffer2);
```

- Inicializar buffers para evitar el uso de datos no terminados.

```
//-- Buffer automático sin inicializar completamente
char data[10];
read(fd, data, sizeof(data));
printf("%s", data);
```

```
//-- Buffer inicializado y terminación controlada
char data2[11] = {0};
ssize_t n = read(fd, data2, sizeof(data2) - 1);
if (n >= 0) {
  data2[n] = '\0';
}
printf("%s", data2);
```

- Validar que funciones externas devuelvan cadenas terminadas en nulo antes de usarlas.

```
//-- Uso directo de datos recibidos sin verificación
char buffer[8];
get_external_data(buffer, sizeof(buffer));
printf("%s", buffer);
```

```
//-- Verificación y aseguramiento de terminación
char buffer2[8];
get_external_data(buffer2, sizeof(buffer2));
```

```
buffer2[sizeof(buffer2) - 1] = '\0';
printf("%s", buffer2);
```

2.1.3.2 Estándares metodológicos

- **CERT/C-STR32**: Do not pass a non-null-terminated character sequence to a library function that expects a string.

2.1.4 [c.str.4] - Convertir caracteres a unsigned char antes de ampliarlos a enteros mayores

ID	c.str.4	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La conversión automática de tipos durante operaciones aritméticas puede causar **comportamientos indefinidos** cuando se trabaja con caracteres que contienen valores negativos. Los caracteres con el bit más significativo activado se interpretan como números negativos al convertirse implícitamente a `int`, produciendo resultados inesperados en funciones de la biblioteca estándar y operaciones de manipulación de bits.

Codificación insegura

Lectura de un carácter desde la entrada estándar y su conversión a un tipo entero para mostrar su valor numérico. Se observa cómo la ampliación directa de un `char` a `int` puede producir valores negativos cuando el tipo `char` es firmado, lo que puede generar errores lógicos o de validación.

```
#include <stdio.h>

int main(void) {
  //-- Declaración de un carácter leído desde la entrada
  char c;

  //-- Solicita al usuario un carácter
  printf("Introduce un caracter: ");
  c = getchar();

  //-- Conversión implícita de char a int
  int valor = c;

  //-- Muestra el valor numérico resultante
  printf("Valor numerico interpretado: %d\n", valor);

  return 0;
```

```
}
```

Codificación recomendada

Lectura de un carácter desde la entrada estándar y conversión segura a un tipo entero mayor. Se fuerza la conversión previa a `unsigned char` antes de ampliarlo a `int`, evitando problemas de extensión de signo y asegurando que el valor numérico esté en el rango esperado de 0 a 255.

```
#include <stdio.h>

int main(void) {
  //-- Declaración de un carácter leído desde la entrada
  char c;

  //-- Solicita al usuario un carácter
  printf("Introduce un caracter: ");
  c = getchar();

  //-- Conversión explícita a unsigned char antes de ampliar a int
  int valor = (unsigned char)c;

  //-- Muestra el valor numérico resultante sin extensión de signo
  printf("Valor numerico interpretado: %d\n", valor);

  return 0;
}
```

2.1.4.1 Descripción

La conversión de caracteres a `unsigned char` antes de su ampliación a tipos enteros mayores previene comportamientos indefinidos durante las operaciones de promoción de tipos. Esta práctica aborda las peculiaridades del tipo `char`, que puede ser `signed` o `unsigned` según la implementación del compilador.

Cuando un `char` con signo contiene valores negativos se amplía automáticamente a un tipo entero mayor, la extensión de signo puede producir resultados inesperados. El bit de signo se propaga a los bits más significativos del tipo destino, generando valores que no corresponden con la representación esperada del carácter original.

Este comportamiento resulta especialmente problemático al trabajar con **funciones de clasificación de caracteres** de la biblioteca estándar, operaciones de manipulación de bits, o cuando se procesan datos binarios interpretados como caracteres. La promoción automática puede introducir bits adicionales no deseados que alteran el resultado de las operaciones.

Los riesgos asociados incluyen **corrupción de datos**, fallos en la validación de entrada, y comportamientos impredecibles en operaciones aritméticas y lógicas. Estos problemas se manifiestan de forma inconsistente entre diferentes arquitecturas y compiladores, dificultando la detección durante las fases de desarrollo y pruebas.

La mitigación se basa en realizar una **conversión explícita** a `unsigned char` antes de cualquier operación que implique promoción a tipos enteros mayores. Esta conversión garantiza que todos los bits del carácter se interpreten como valores positivos, eliminando la extensión de signo no deseada. La aplicación sistemática de esta práctica en interfaces con funciones de biblioteca y operaciones de manipulación de datos asegura un comportamiento consistente y predecible del código.

Riesgos asociados

- Lectura incorrecta de caracteres con valor negativo, produciendo resultados inesperados al convertir a tipos enteros mayores.
- Comparaciones erróneas al tratar caracteres con signo como enteros, provocando ramas de ejecución inadecuadas.
- Vulnerabilidades en funciones de biblioteca estándar (como `isdigit()` o `isalpha()`) si se pasan valores fuera del rango `unsigned char`, lo que puede llevar a un comportamiento indefinido.
- Errores de manipulación de datos binarios que usan el tipo `char`, introduciendo posibles fallos lógicos o corrupciones de datos.
- Exposición a explotaciones de seguridad, como desbordamientos de buffer, por resultados inesperados en cálculos o comparaciones derivados del signo incorrecto.
- Generación de registros y logs incorrectos que dificulten la auditoría y el rastreo de incidentes.

Buenas prácticas

- Realizar un casting explícito a `unsigned char` antes de promover un `char` a un tipo entero mayor para evitar extensiones de signo inesperadas.

```
//-- Promoción directa de char con posible extensión de signo
char c = getc(f);
int valor = c;
```

```
//-- Conversión explícita a unsigned char antes de promover
char c = getc(f);
int valor = (unsigned char)c;
```

- Convertir a `unsigned char` antes de pasar caracteres a funciones de la familia

ctype.h.

```
//-- Paso directo de char potencialmente negativo a isalpha
char c = buffer[i];
if (isalpha(c)) {
  procesar();
}
```

```
//-- Conversión a unsigned char antes de llamar a isalpha
char c = buffer[i];
if (isalpha((unsigned char)c)) {
  procesar();
}
```

- Aplicar la conversión a `unsigned char` antes de realizar operaciones aritméticas o comparaciones que impliquen promoción implícita.

```
//-- Comparación con promoción implícita que puede alterar el valor
char c = entrada;
if (c > 200) {
  alerta();
}
```

```
//-- Conversión previa a unsigned char para preservar el valor real
char c = entrada;
if ((unsigned char)c > 200) {
  alerta();
}
```

2.1.4.2 Estándares metodológicos

- **CERT/C-STR34**: Cast characters to unsigned char before converting to larger integer sizes.

2.1.5 [c.str.5] - Garantizar que los argumentos a funciones de caracteres sean representables como unsigned char

ID	c.str.5	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de la biblioteca estándar que procesan caracteres pueden tener un **comportamiento inesperado** cuando reciben argumentos fuera del rango válido. Estos argumentos deben ser representables como `unsigned char` o tener el valor `EOF`,

de lo contrario pueden producirse fallos de segmentación, corrupción de datos o resultados impredecibles en el programa.

Codificación insegura

Uso de funciones de la biblioteca <ctype.h> con valores de tipo `char` que pueden ser negativos. Se busca validar si un carácter es alfabético a partir de la entrada del usuario, mostrando el impacto de pasar directamente un `char` con signo a funciones como `isalpha()`, lo que puede provocar comportamiento indefinido si el valor no es representable como `unsigned char`.

```
#include <stdio.h>
#include <ctype.h>

//-- Ejemplo de uso directo de char en funciones de ctype
int main(void) {
  //-- Variable tipo char que puede ser signed dependiendo del compilador
  char c;

  //-- Solicita un carácter al usuario
  printf("Introduce un caracter: ");
  c = getchar();

  //-- Uso directo de isalpha con char potencialmente negativo
  if (isalpha(c)) {
    //-- Se considera letra
    printf("Es una letra\n");
  } else {
    //-- No se considera letra
    printf("No es una letra\n");
  }
  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Validación segura del carácter antes de pasarlo a funciones de <ctype.h>, asegurando que el argumento sea representable como `unsigned char`. Se aplica una conversión explícita para evitar comportamiento indefinido cuando el valor original es negativo.

```
#include <stdio.h>
#include <ctype.h>

//-- Ejemplo de conversión explícita a unsigned char antes de usar ctype
int main(void) {
  //-- Variable tipo char que puede ser signed dependiendo del compilador
  char c;
```

```
  //-- Solicita un carácter al usuario
  printf("Introduce un caracter: ");
  c = getchar();

  //-- Conversión segura a unsigned char antes de llamar a isalpha
  if (isalpha((unsigned char)c)) {
    //-- Se considera letra
    printf("Es una letra\n");
  } else {
    //-- No se considera letra
    printf("No es una letra\n");
  }
  //-- Fin del programa
  return 0;
}
```

2.1.5.1 Descripción

El problema se origina cuando se pasan valores de tipo `char` con signo a funciones como `isalpha`, `isdigit`, `toupper` o `tolower`. En sistemas donde `char` tiene signo por defecto, los caracteres con valores negativos pueden causar **comportamiento indefinido** al ser interpretados incorrectamente por estas funciones.

Los caracteres extendidos del conjunto ASCII, aquellos con valores entre 128 y 255, son particularmente problemáticos. Cuando se almacenan en variables `char` con signo, estos valores se interpretan como números negativos, lo que viola las precondiciones de las funciones de caracteres.

Este comportamiento indefinido puede manifestarse de diferentes formas según la implementación: desde resultados incorrectos hasta fallos de segmentación. La **portabilidad del código** también se ve comprometida, ya que el comportamiento puede variar entre diferentes compiladores y arquitecturas.

La mitigación principal consiste en realizar una **conversión explícita** a `unsigned char` antes de pasar el argumento a las funciones de caracteres. Esta conversión garantiza que el valor esté dentro del rango esperado y elimina la dependencia del signo del tipo `char` en la plataforma específica.

Alternativamente, se puede verificar que el valor del carácter sea no negativo antes de pasarlo a estas funciones, aunque la conversión explícita resulta más robusta y clara en términos de intención del código.

Riesgos asociados

- Comportamiento indefinido al pasar valores fuera del rango de `unsigned char`, lo que puede causar fallos o resultados impredecibles.

- Errores de acceso a memoria por desbordamiento de buffer al interpretar caracteres con signo como índices o longitudes negativas.
- Errores lógicos debidos a la interpretación incorrecta de valores negativos, especialmente en funciones estándar como `isalpha()`, `isdigit()`, etc.
- Potenciales vulnerabilidades de seguridad explotables, como ejecución de código arbitrario o denegación de servicio.
- Dificultad para depurar y mantener el código, ya que los efectos de pasar un valor fuera del rango válido pueden no ser evidentes de inmediato.
- Posibles problemas de portabilidad entre distintas implementaciones o plataformas debido a diferencias en la representación de `char`.

Buenas prácticas

- Convertir explícitamente a `unsigned char` antes de pasar argumentos a funciones de `<ctype.h>`.

```
//-- Uso directo de char potencialmente negativo
char c = buffer[i];
if (isalpha(c)) {
  procesar();
}
```

```
//-- Conversión explícita a unsigned char antes de la llamada
char c = buffer[i];
if (isalpha((unsigned char)c)) {
  procesar();
}
```

- Evitar pasar valores `char` con signo provenientes de entrada externa sin validación previa.

```
//-- Paso directo de dato externo a función de caracteres
char input = getc(f);
if (toupper(input) == 'A') {
  accion();
}
```

```
//-- Validación y conversión segura antes de usar la función
int input = getc(f);
if (input != EOF && toupper((unsigned char)input) == 'A') {
  accion();
}
```

- Definir variables intermedias como `unsigned char` cuando se procesen datos

binarios o extendidos.

```
//-- Uso de char con posible extensión de signo
char byte = datos[pos];
if (isspace(byte)) {
  limpiar();
}
```

```
//-- Declaración explícita como unsigned char para evitar valores negativos
unsigned char byte = datos[pos];
if (isspace(byte)) {
  limpiar();
}
```

- No asumir que `char` es unsigned por defecto; realizar conversión explícita en expresiones.

```
//-- Expresión directa con char firmado
char c = texto[j];
if (isdigit(c)) {
  contador++;
}
```

```
//-- Conversión explícita dentro de la expresión condicional
char c = texto[j];
if (isdigit((unsigned char)texto[j])) {
  contador++;
}
```

2.1.5.2 Estándares metodológicos

- **CERT/C-STR37**: Arguments to character-handling functions must be representable as an unsigned char.

2.1.6 [c.str.6] - No confundir cadenas y funciones de caracteres estrechos y anchos

ID	c.str.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de manipulación de cadenas y caracteres se dividen en dos categorías: estrechas (narrow) y anchas (wide), cada una diseñada para tipos de datos específicos. El uso incorrecto de estas funciones genera **corrupción de datos** y vulnerabilidades de

seguridad. Los errores más frecuentes incluyen pasar cadenas `wchar_t` a funciones como `strlen` o usar `wcslen` con cadenas `char`.

Codificación insegura

Uso incorrecto de funciones de caracteres anchos sobre cadenas de caracteres estrechos. Se busca mostrar cómo la conversión forzada entre tipos puede compilar, pero produce comportamiento indefinido al mezclar `char` con funciones diseñadas para `wchar_t`, generando resultados erróneos o fallos en ejecución.

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>

int main(void) {
  //-- Cadena de caracteres estrechos (char)
  char texto[] = "Hola mundo";

  //-- Uso incorrecto: conversión forzada de char* a wchar_t*
  //-- Esto compila, pero produce comportamiento indefinido
  size_t longitud = wcslen((wchar_t*)texto);

  //-- Se imprime el resultado, que puede ser incorrecto
  printf("Longitud calculada: %zu\n", longitud);

  return 0;
}
```

Codificación recomendada

Uso correcto y consistente de cadenas y funciones de caracteres anchos. Se muestra cómo declarar una cadena `wchar_t`, configurar la localización y utilizar funciones específicas para caracteres anchos sin mezclar tipos, garantizando comportamiento definido y portable.

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main(void) {
  //-- Configura la localización para soportar caracteres anchos
  setlocale(LC_ALL, "");

  //-- Cadena de caracteres anchos (wchar_t)
  wchar_t texto[] = L"Hola mundo";

  //-- Uso correcto de función para caracteres anchos
  size_t longitud = wcslen(texto);
```

```
    //-- Impresión usando función específica para wchar_t
    wprintf(L"Longitud calculada: %zu\n", longitud);

    return 0;
}
```

2.1.6.1 Descripción

Los caracteres estrechos utilizan el tipo `char` y sus cadenas se representan como `char*`. Las funciones asociadas incluyen `strlen`, `strcpy`, `strcmp` y otras de la familia `str*`. Por otro lado, los caracteres anchos emplean el tipo `wchar_t` y sus cadenas como `wchar_t*`, con funciones específicas como `wcslen`, `wcscpy`, `wcscmp` y la familia `wcs*`.

La confusión entre estos dos tipos de funciones genera **comportamientos indefinidos** y errores de ejecución. El uso incorrecto puede provocar corrupción de memoria, lecturas fuera de límites y fallos en el procesamiento de datos, especialmente cuando se manejan caracteres multibyte o codificaciones internacionales.

Los compiladores no siempre detectan estos errores de tipo durante la compilación, ya que ambos tipos pueden convertirse implícitamente a punteros genéricos. Esta característica del lenguaje aumenta la probabilidad de errores silenciosos que se manifiestan únicamente durante la ejecución.

La mitigación de estos problemas requiere **verificación consistente de tipos** y el uso exclusivo de funciones apropiadas para cada tipo de carácter. Las herramientas de análisis estático pueden detectar muchos de estos errores durante el desarrollo. Además, la activación de advertencias del compilador relacionadas con conversiones de tipos ayuda a identificar posibles inconsistencias antes de la ejecución.

Riesgos asociados

- Desbordamiento de buffer, al interpretar incorrectamente el tamaño de la cadena.
- Lectura/escritura fuera de límites de memoria.
- Condiciones de corrupción de memoria y datos.
- Fallos de segmentación (segmentation faults) y caídas del programa.
- Exposición a vulnerabilidades de seguridad como ejecución de código arbitrario.
- Resultados inesperados en operaciones de comparación y manipulación de cadenas.
- Problemas de portabilidad y compatibilidad entre diferentes plataformas.
- Divulgación accidental de información sensible o confidencial.

- Problemas en la internacionalización y localización de la aplicación.

Buenas prácticas

- No mezclar funciones de manejo de cadenas estrechas (`char`) con cadenas anchas (`wchar_t`).

```
//-- Uso de strlen con cadena ancha
wchar_t *ws = L"Hola";
size_t len = strlen(ws);
```

```
//-- Uso de wcslen con cadena ancha
wchar_t *ws2 = L"Hola";
size_t len2 = wcslen(ws2);
```

- Utilizar literales correctos (`"texto"` vs `L"texto"`) según el tipo de dato declarado.

```
//-- Asignación de literal estrecho a puntero ancho
wchar_t *ws = "Hola";
```

```
//-- Asignación de literal ancho compatible
wchar_t *ws2 = L"Hola";
```

- No usar funciones de copia estrechas para buffers anchos.

```
//-- Copia con strcpy en buffer wchar_t
wchar_t buffer[20];
strcpy((char *)buffer, "Hola");
```

```
//-- Copia con wcscpy en buffer wchar_t
wchar_t buffer2[20];
wcscpy(buffer2, L"Hola");
```

- Evitar comparar cadenas anchas con funciones de comparación estrechas.

```
//-- Comparación con strcmp sobre wchar_t
wchar_t *ws = L"Hola";
if (strcmp((char *)ws, "Hola") == 0) {
  /* ... */
}
```

```
//-- Comparación con wcscmp sobre wchar_t
wchar_t *ws2 = L"Hola";
if (wcscmp(ws2, L"Hola") == 0) {
  /* ... */
```

```
}
```

- Mantener coherencia en funciones de entrada/salida para caracteres estrechos y anchos.

```
//-- Impresión con printf de cadena ancha
wchar_t *ws = L"Hola";
printf("%s", ws);
```

```
//-- Impresión con wprintf de cadena ancha
wchar_t *ws2 = L"Hola";
wprintf(L"%ls", ws2);
```

2.1.6.2 Estándares metodológicos

- **CERT/C-STR38**: Do not confuse narrow and wide character strings and functions.

2.2 Conversiones

2.2.1 [c.conv.1] - Evitar wraparound en operaciones con enteros sin signo

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

Las operaciones aritméticas con enteros sin signo pueden producir **wraparound** cuando el resultado excede el valor máximo representable del tipo de dato. Este comportamiento, aunque está definido en el estándar, genera resultados inesperados que comprometen la lógica del programa y pueden introducir vulnerabilidades de seguridad.

Codificación insegura

Se muestra una operación aritmética con enteros sin signo donde se realiza una resta sin validar los límites. El objetivo es observar cómo una operación aparentemente válida puede provocar un desbordamiento por wraparound cuando el resultado esperado sería negativo, generando un valor muy grande e inesperado.

```
#include <stdio.h>
#include <limits.h>

int main(void) {
  //-- Declaración de variables sin signo
```

```
  unsigned int saldo = 10;
  unsigned int retiro = 20;

  //-- Resta sin validación previa
  unsigned int resultado = saldo - retiro;

  //-- Se imprime el resultado afectado por wraparound
  printf("Saldo inicial: %u\n", saldo);
  printf("Retiro: %u\n", retiro);
  printf("Resultado despues de la resta: %u\n", resultado);

  return 0;
}
```

Codificación recomendada

Se presenta una resta entre enteros sin signo validando previamente que la operación no provoque un wraparound. El objetivo es aprender a comprobar condiciones antes de realizar operaciones aritméticas que podrían producir resultados inesperados cuando se trabaja con tipos sin signo.

```
#include <stdio.h>
#include <limits.h>

int main(void) {
  //-- Declaración de variables sin signo
  unsigned int saldo = 10;
  unsigned int retiro = 20;

  //-- Validación para evitar wraparound
  if (retiro > saldo) {
    //-- Se evita la operación insegura
    printf("Error: el retiro es mayor que el saldo disponible\n");
  } else {
    //-- Operación segura tras validación
    unsigned int resultado = saldo - retiro;
    printf("Resultado despues de la resta: %u\n", resultado);
  }
  return 0;
}
```

2.2.1.1 Descripción

El wraparound ocurre cuando una operación aritmética produce un resultado que supera `UINT_MAX`, `ULONG_MAX` o el límite correspondiente del tipo utilizado. En estos casos, el valor almacenado no es el resultado matemático completo, sino el resto de dividir ese resultado entre la cantidad de valores representables por el tipo. Por ejemplo, si un tipo

sin signo representa valores de 0 a UINT_MAX, entonces UINT_MAX + 1 produce 0, UINT_MAX + 2 produce 1, y así sucesivamente.

Este fenómeno representa un riesgo considerable en contextos donde los valores se utilizan para **asignación de memoria**, **control de bucles**, **validación de límites** o **cálculos de índices de arrays**. Los resultados incorrectos pueden provocar accesos a memoria no válida, desbordamientos de buffer o comportamientos impredecibles del programa.

La mitigación de este problema requiere **validación previa** de los operandos antes de realizar las operaciones aritméticas. Es necesario verificar que el resultado de la operación no excederá los límites del tipo de dato utilizado. Las bibliotecas estándar proporcionan constantes como `UINT_MAX` que facilitan estas comprobaciones.

Adicionalmente, el uso de tipos de datos con mayor capacidad para operaciones intermedias y la implementación de **funciones de verificación de rango** constituyen estrategias efectivas para prevenir el wraparound en operaciones críticas.

Riesgos asociados

- Desbordamiento de enteros, causando cálculos incorrectos o inesperados.
- Vulnerabilidades de seguridad por permitir que se eludan validaciones de límites.
- Posibilidad de condiciones fuera de límites (out-of-bounds) en accesos a memoria o arrays.
- Ejecución no controlada de bucles o estructuras de control debido a contadores incorrectos.
- Bypass de comprobaciones de tamaño, permitiendo ataques como buffer overflow o integer overflow.
- Fallo en lógicas críticas de seguridad que dependen de valores correctos de enteros.
- Bugs difíciles de detectar y reproducir durante pruebas o auditorías de código.
- Compromiso de la integridad de los datos manipulados por el programa.

Buenas prácticas

- Validar explícitamente los límites antes de realizar sumas con enteros sin signo.

```
//-- Suma sin validar puede provocar wraparound
unsigned int total = a + b;
```

```
//-- Validación previa contra el valor máximo antes de sumar
if (a > UINT_MAX - b) {
  /* manejar error */
}
```

```
unsigned int total = a + b;
```

- Comprobar underflow antes de realizar restas con tipos sin signo.

```
//-- Resta directa puede generar wraparound si b > a
unsigned int resultado = a - b;
```

```
//-- Verificación previa para evitar underflow
if (a < b) {
  /* manejar error */
}
unsigned int resultado = a - b;
```

- Validar multiplicaciones comprobando posibles desbordamientos antes de operar.

```
//-- Multiplicación sin control puede desbordar
unsigned int area = width * height;
```

```
//-- Comprobación previa usando división inversa
if (width != 0 && height > UINT_MAX / width) {
  /* manejar error */
}
unsigned int area = width * height;
```

- Evitar conversiones implícitas que puedan ocultar wraparound entre tipos de distinto tamaño.

```
//-- Conversión implícita puede truncar el valor
unsigned int len = get_size();
unsigned short small = len;
```

```
//-- Validación explícita antes de convertir
unsigned int len = get_size();
if (len > USHRT_MAX) {
  /* manejar error */
}
unsigned short small = (unsigned short)len;
```

- Validar tamaños calculados antes de utilizarlos en asignaciones de memoria.

```
//-- Cálculo de tamaño sin validar puede causar wraparound
unsigned int size = count * sizeof(struct item);
void *ptr = malloc(size);
```

```
//-- Comprobación previa antes de reservar memoria
if (count != 0 && sizeof(struct item) > UINT_MAX / count) {
  /* manejar error */
}
unsigned int size = count * sizeof(struct item);
void *ptr = malloc(size);
```

2.2.1.2 Estándares metodológicos

- **CERT/C-INT30**: Ensure that unsigned integer operations do not wrap.

2.2.2 [c.conv.2] - Evitar pérdida o mala interpretación de datos en conversiones entre enteros

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

Las conversiones entre tipos enteros pueden provocar **pérdida de datos** o **interpretación incorrecta** de valores cuando se realizan entre tipos de diferente tamaño o signo. Estos problemas surgen al convertir desde tipos más grandes a más pequeños, o entre tipos con y sin signo, generando comportamientos impredecibles que comprometen la integridad de los datos y pueden introducir vulnerabilidades de seguridad en el programa.

Codificación insegura

Conversión de un entero de mayor tamaño a uno de menor tamaño sin validar el rango, mostrando cómo puede producirse truncamiento y pérdida de información cuando el valor excede la capacidad del tipo destino. Se busca observar cómo el resultado cambia inesperadamente debido a una conversión implícita.

```
#include <stdio.h>
#include <stdint.h>

//-- Programa que muestra pérdida de datos por conversión implícita
int main() {
  //-- Declaración de un entero de 32 bits con un valor grande
  int32_t valorGrande = 70000;

  //-- Conversión implícita a entero de 16 bits
  int16_t valorPequeno = valorGrande;

  //-- Impresión de ambos valores
  printf("Valor original (int32_t): %d\n", valorGrande);
  printf("Valor convertido (int16_t): %d\n", valorPequeno);
```

```
  return 0;
}
```

Codificación recomendada

Conversión controlada entre enteros de distinto tamaño validando previamente que el valor se encuentra dentro del rango permitido por el tipo destino. Se pretende evitar truncamientos y asegurar que la conversión solo se realiza cuando es segura.

```
#include <stdio.h>
#include <stdint.h>
#include <limits.h>

//-- Programa que valida el rango antes de convertir
int main() {
  //-- Declaración de un entero de 32 bits con un valor grande
  int32_t valorGrande = 70000;

  //-- Verificación de que el valor cabe en int16_t
  if (valorGrande < INT16_MIN || valorGrande > INT16_MAX) {
    //-- Mensaje de error si el valor está fuera de rango
    printf("Error: el valor %d está fuera del rango de int16_t\n", valorGrande);
    return 1;
  }
  //-- Conversión segura después de validar el rango
  int16_t valorPequeno = (int16_t)valorGrande;

  //-- Impresión del valor convertido correctamente
  printf("Valor convertido correctamente (int16_t): %d\n", valorPequeno);

  return 0;
}
```

2.2.2.1 Descripción

Los problemas de conversión surgen principalmente en tres escenarios: conversiones entre tipos con diferente signo, conversiones a tipos con menor rango de representación, y conversiones que involucran promociones enteras automáticas. El comportamiento puede ser **implementation-defined** o **undefined** según el estándar, lo que compromete la portabilidad y predictibilidad del código.

Las conversiones problemáticas incluyen el truncamiento de valores que exceden el rango del tipo destino, la reinterpretación de valores con signo como valores sin signo y viceversa, y las conversiones implícitas durante operaciones aritméticas que pueden alterar el resultado esperado.

Los riesgos asociados abarcan desde cálculos incorrectos y lógica de control alterada hasta vulnerabilidades de seguridad como desbordamientos de buffer y bypass de validaciones. La pérdida silenciosa de datos puede introducir errores difíciles de detectar durante las pruebas.

La mitigación requiere validación explícita de rangos antes de realizar conversiones, uso de tipos apropiados para cada contexto, y verificación de que los valores se mantienen dentro de los límites esperados. Las herramientas de análisis estático pueden detectar muchas conversiones potencialmente problemáticas durante el desarrollo.

Riesgos asociados

- Pérdida de información significativa al truncar valores durante conversiones entre tipos enteros grandes y pequeños.
- Desbordamientos o subdesbordamientos al representar valores fuera del rango del tipo destino.
- Comportamiento indefinido por conversiones de enteros con signo a enteros sin signo y viceversa.
- Vulnerabilidades de seguridad, como ejecución de código no esperado al usar enteros mal convertidos como índices, tamaños de memoria o contadores de bucles.
- Errores lógicos y fallos en validaciones, condicionales o comparaciones numéricas, provocando ejecución incorrecta del programa.
- Generación de resultados corruptos y posibilidad de fugas de información sensible por manejo incorrecto de los datos.
- Incompatibilidad entre plataformas por diferencias en la representación y el rango de los tipos de datos enteros.

Buenas prácticas

- Validar explícitamente los límites antes de convertir entre enteros de distinto tamaño.

```
//-- Conversión directa sin validar rango
int32_t valor = obtener_valor();
int16_t reducido = valor;
```

```
//-- Validación explícita de límites antes de convertir
int32_t valor = obtener_valor();
if (valor >= INT16_MIN && valor <= INT16_MAX) {
  int16_t reducido = (int16_t)valor;
}
```

- Evitar conversiones implícitas entre tipos con distinto signo.

```
//-- Comparación entre entero con signo y sin signo
int32_t a = -1;
uint32_t b = 1;
if (a < b) {
  procesar();
}
```

```
//-- Conversión explícita y validación previa del signo
int32_t a = -1;
uint32_t b = 1;
if (a >= 0 && (uint32_t)a < b) {
  procesar();
}
```

- Utilizar tipos de tamaño fijo para evitar ambigüedad en plataformas distintas.

```
//-- Uso de tipos dependientes de la implementación
long dato = obtener_dato();
int resultado = dato;
```

```
//-- Uso de tipos de tamaño fijo y conversión controlada
int32_t dato = obtener_dato();
if (dato >= INT16_MIN && dato <= INT16_MAX) {
  int16_t resultado = (int16_t)dato;
}
```

- Comprobar posibles desbordamientos antes de realizar operaciones que cambien el rango.

```
//-- Multiplicación que puede desbordar antes de convertir
int32_t x = obtener_valor();
int16_t y = x * 2;
```

```
//-- Verificación previa del rango antes de operar y convertir
int32_t x = obtener_valor();
if (x <= INT16_MAX / 2 && x >= INT16_MIN / 2) {
  int16_t y = (int16_t)(x * 2);
}
```

- Evitar truncamientos involuntarios al convertir a tipos de menor tamaño.

```
//-- Conversión que trunca bits más significativos
uint32_t valor = 0x12345678;
```

```
uint8_t byte = valor;
```

```
//-- Enmascarado explícito para indicar la intención de truncamiento
uint32_t valor = 0x12345678;
uint8_t byte = (uint8_t)(valor & 0xFF);
```

2.2.2.2 Estándares metodológicos

- **CERT/C-INT31**: Ensure that integer conversions do not result in lost or misinterpreted data.

2.2.3 [c.conv.3] - Evitar desbordamientos en operaciones con enteros con signo

ID	c.conv.3	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los desbordamientos en operaciones aritméticas con enteros con signo producen **resultados inesperados** que puede comprometer la seguridad y estabilidad del programa. Estas situaciones ocurren cuando el resultado de operaciones como suma, resta, multiplicación o negación excede los límites representables del tipo de dato, generando resultados impredecibles que varían según la implementación del compilador y la arquitectura del sistema.

Codificación insegura

Operación de suma entre dos enteros con signo recibidos por entrada estándar. Se muestra cómo una suma directa puede producir un desbordamiento cuando los valores superan los límites del tipo `int`, y cómo controlar previamente los límites para evitar resultados incorrectos o comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Declaración de variables para almacenar los operandos
  int a;
  int b;
  int resultado;

  //-- Solicita al usuario dos números enteros
  printf("Introduce el primer numero: ");
  if (scanf("%d", &a) != 1) {
    printf("Entrada invalida\n");
```

```
    return EXIT_FAILURE;
  }
  printf("Introduce el segundo numero: ");
  if (scanf("%d", &b) != 1) {
    printf("Entrada invalida\n");
    return EXIT_FAILURE;
  }
  //-- Suma directa sin comprobar posibles desbordamientos
  resultado = a + b;

  //-- Muestra el resultado que puede ser incorrecto si hubo overflow
  printf("Resultado: %d\n", resultado);

  return EXIT_SUCCESS;
}
```

Codificación recomendada

Operación de suma entre dos enteros con signo validando previamente que la operación no exceda los límites definidos por el tipo `int`. Se utilizan las constantes de `limits.h` para comprobar si la suma provocaría un desbordamiento antes de realizarla.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main(void) {
  //-- Declaración de variables para almacenar los operandos
  int a;
  int b;
  int resultado;

  //-- Solicita al usuario dos números enteros
  printf("Introduce el primer numero: ");
  if (scanf("%d", &a) != 1) {
    printf("Entrada invalida\n");
    return EXIT_FAILURE;
  }
  printf("Introduce el segundo numero: ");
  if (scanf("%d", &b) != 1) {
    printf("Entrada invalida\n");
    return EXIT_FAILURE;
  }
  //-- Verifica si la suma provocaría un desbordamiento positivo o negativo
  if ((b > 0 && a > INT_MAX - b) ||
    (b < 0 && a < INT_MIN - b)) {
    //-- Notifica el error y evita realizar la operación insegura
    printf("Error: la suma provocaria desbordamiento\n");
```

```
    return EXIT_FAILURE;
  }
  //-- Realiza la suma solo cuando es segura
  resultado = a + b;

  //-- Muestra el resultado validado
  printf("Resultado: %d\n", resultado);

  return EXIT_SUCCESS;
}
```

2.2.3.1 Descripción

Los desbordamientos en operaciones con enteros con signo constituyen un comportamiento indefinido que puede **comprometer la integridad y seguridad del programa**. Cuando una operación aritmética produce un resultado que excede los límites representables del tipo de dato, el comportamiento resultante no está especificado por el estándar y puede variar entre compiladores y arquitecturas.

Este tipo de desbordamiento puede ocurrir en operaciones básicas como suma, resta, multiplicación y división, así como en conversiones entre tipos de diferentes tamaños. El comportamiento indefinido resultante puede manifestarse de diversas formas: desde resultados incorrectos hasta terminación inesperada del programa o vulnerabilidades de seguridad explotables.

Los riesgos asociados incluyen corrupción de datos, ejecución de código malicioso, bypass de validaciones de seguridad y comportamiento impredecible del programa. Estos problemas son especialmente críticos en aplicaciones que manejan datos sensibles o controlan sistemas críticos.

La mitigación efectiva requiere implementar **verificaciones previas a las operaciones aritméticas** para detectar condiciones que puedan causar desbordamiento. Las estrategias incluyen el uso de tipos de datos de mayor capacidad para operaciones intermedias, la implementación de funciones de verificación de límites y el empleo de bibliotecas especializadas que proporcionan operaciones aritméticas seguras.

Los compiladores modernos ofrecen opciones de detección de desbordamientos y bibliotecas de funciones que realizan verificaciones automáticas. La validación de entrada de datos y el establecimiento de rangos válidos para las operaciones también contribuyen a prevenir estas situaciones.

Riesgos asociados

- Ejecutar instrucciones o rutas de código inesperadas debido a resultados incorrectos en comparaciones o cálculos.
- Vulnerabilidad a ataques de desbordamiento de buffer cuando los valores

enteros se usan como tamaños de memoria o índices de arrays.

- Comportamiento indefinido del programa, llevando a fallos de ejecución, bloqueos o corrupción de datos.
- Escalado de privilegios o ejecución de código arbitrario si un atacante explota el desbordamiento para manipular el flujo de control.
- Filtración accidental de información sensible a través de errores lógicos o sobrescritura de memoria.
- Resultados incorrectos en cálculos financieros, científicos o de negocio, con impacto potencial en la integridad de la aplicación.
- Dificultad para detectar y depurar errores debido a la naturaleza silenciosa de los desbordamientos en C.

Buenas prácticas

- Validar explícitamente los límites antes de realizar operaciones aritméticas con enteros con signo.

```
//-- Suma sin validar posibles límites
int a = INT_MAX;
int b = 1;
int c = a + b;
```

```
//-- Validación previa contra el límite superior
int a = INT_MAX;
int b = 1;
if (a > INT_MAX - b) {
  /* manejar error */
}
int c = a + b;
```

- Comprobar límites antes de realizar multiplicaciones que puedan exceder la capacidad del tipo.

```
//-- Multiplicación sin comprobación de rango
int x = 50000;
int y = 50000;
int z = x * y;
```

```
//-- Validación previa usando división inversa para evitar desbordamiento
int x = 50000;
int y = 50000;
if (x != 0 && y > INT_MAX / x) {
  /* manejar error */
```

```
}
int z = x * y;
```

- Evitar restas que puedan producir desbordamiento verificando límites mínimos.

```
//-- Resta sin validación de límite inferior
int a = INT_MIN;
int b = 1;
int c = a - b;
```

```
//-- Validación previa contra el límite inferior
int a = INT_MIN;
int b = 1;
if (a < INT_MIN + b) {
  /* manejar error */
}
int c = a - b;
```

- Utilizar tipos de mayor capacidad cuando el rango esperado pueda superar el de `int`.

```
//-- Uso de tipo con rango insuficiente
int total = cantidad * precio;
```

```
//-- Uso de tipo con mayor capacidad para la operación
long long total = (long long)cantidad * precio;
```

- Utilizar funciones o utilidades seguras que detecten desbordamientos aritméticos.

```
//-- Operación directa sin detección de desbordamiento
int resultado = a + b;
```

```
//-- Uso de función integrada para detectar desbordamiento
int resultado;
if (__builtin_add_overflow(a, b, &resultado)) {
  /* manejar error */
}
```

2.2.3.2 Estándares metodológicos

- **CERT/C-INT32**: Ensure that operations on signed integers do not result in overflow.

2.2.4 [c.conv.4] - Evitar división por cero en divisiones enteras

ID	c.conv.4	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las operaciones de división y módulo, si no se validan adecuadamente, podrían ejecutar una división por cero o un módulo de 0. Este problema genera fallos de ejecución que van desde una terminación abrupta del programa hasta la corrupción de datos en memoria.

Codificación insegura

Se muestra un programa que realiza una división entera entre dos números introducidos por el usuario. El objetivo es evidenciar qué ocurre cuando no se valida el divisor antes de ejecutar la operación y cómo puede producirse un fallo en tiempo de ejecución.

```
#include <stdio.h>

//-- Programa que realiza una división entera sin validar el divisor
int main() {
  int numerador;
  int divisor;
  int resultado;

  //-- Solicita el numerador al usuario
  printf("Ingrese el numerador: ");
  scanf("%d", &numerador);

  //-- Solicita el divisor al usuario
  printf("Ingrese el divisor: ");
  scanf("%d", &divisor);

  //-- Realiza la division sin comprobar si el divisor es cero
  resultado = numerador / divisor;

  //-- Muestra el resultado de la division
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

Codificación recomendada

Se presenta un programa que realiza una división entera validando previamente que el divisor no sea cero. Se pretende mostrar cómo aplicar una comprobación sencilla para evitar errores de ejecución y comportamientos indefinidos.

```c
#include <stdio.h>

//-- Programa que valida el divisor antes de realizar la division
int main() {
  int numerador;
  int divisor;
  int resultado;

  //-- Solicita el numerador al usuario
  printf("Ingrese el numerador: ");
  if (scanf("%d", &numerador) != 1) {
    //-- Error en la lectura del numerador
    printf("Entrada invalida.\n");
    return 1;
  }
  //-- Solicita el divisor al usuario
  printf("Ingrese el divisor: ");
  if (scanf("%d", &divisor) != 1) {
    //-- Error en la lectura del divisor
    printf("Entrada invalida.\n");
    return 1;
  }
  //-- Verifica que el divisor no sea cero antes de dividir
  if (divisor == 0) {
    //-- Evita la division por cero y finaliza el programa
    printf("Error: no se puede dividir por cero.\n");
    return 1;
  }
  //-- Realiza la division de forma segura
  resultado = numerador / divisor;

  //-- Muestra el resultado de la division
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

2.2.4.1 Descripción

La división por cero en operaciones con enteros constituye un comportamiento indefinido que puede provocar la terminación abrupta del programa o generar resultados impredecibles. Este problema surge cuando el divisor en una operación de división o módulo tiene valor cero, situación que el procesador no puede resolver matemáticamente.

El comportamiento indefinido asociado a esta condición puede manifestarse de diferentes formas según la plataforma y el compilador utilizado. Algunos sistemas generan excepciones que terminan la ejecución del programa, mientras que otros

pueden producir valores arbitrarios o corromper el estado del programa de manera silenciosa.

Los riesgos derivados de no validar los divisores incluyen **fallos de disponibilidad** por terminación inesperada del programa, **comportamientos impredecibles** que comprometen la lógica de la aplicación, y potenciales **vulnerabilidades de seguridad** cuando los atacantes pueden controlar los valores de entrada para provocar estas condiciones.

La mitigación de este problema requiere la **validación explícita** de los valores del divisor antes de realizar operaciones de división o módulo. Esta validación debe implementarse tanto para divisiones directas como para operaciones de módulo mediante el operador `%`. Las comprobaciones deben considerar no solo valores literales cero, sino también variables y expresiones que puedan evaluarse a cero durante la ejecución.

La detección de divisores cero debe integrarse en la lógica de manejo de errores del programa, permitiendo una respuesta controlada ante estas situaciones excepcionales. Esta aproximación garantiza que el programa mantenga un comportamiento predecible y controlado incluso cuando se encuentren condiciones de división por cero.

Riesgos asociados

- Fallo de ejecución (crash) de la aplicación por excepción de división por cero.
- Comportamiento indefinido del programa, lo que puede provocar errores difíciles de depurar.
- Vulnerabilidad de denegación de servicio (DoS) al permitir que un atacante provoque caídas intencionadas de la aplicación.
- Exposición a ataques de corrupción de memoria si no se gestiona correctamente la excepción.
- Pérdida de integridad de los datos o de resultados de cálculo inexactos o inesperados.
- Posible bypass de controles de seguridad o lógica si la excepción es atrapada de forma insegura.

Buenas prácticas

- Validar explícitamente que el divisor sea distinto de cero antes de realizar una división entera.

```
//-- División sin validar el divisor
int resultado = total / divisor;
```

```
//-- Validación previa del divisor antes de dividir
if (divisor != 0) {
```

```
  int resultado = total / divisor;
}
```

- Incorporar manejo de errores cuando el divisor pueda provenir de entrada externa o no confiable.

```
//-- Uso directo de valor externo sin comprobación
int divisor = atoi(buffer);
int promedio = suma / divisor;
```

```
//-- Validación y manejo de error ante divisor inválido
int divisor = atoi(buffer);
if (divisor == 0) {
  fprintf(stderr, "Error: divisor invalido\n");
  return;
}
int promedio = suma / divisor;
```

- Inicializar variables divisor con valores seguros y controlados para evitar estados no definidos.

```
//-- Variable divisor sin inicializar
int divisor;
int resultado = valor / divisor;
```

```
//-- Inicialización segura y verificación posterior
int divisor = 1;
if (condicion) {
  divisor = obtener_divisor();
}
if (divisor != 0) {
  int resultado = valor / divisor;
}
```

2.2.4.2 Estándares metodológicos

- **CERT/C-INT33**: Ensure that division and remainder operations do not result in divide-by-zero errors.

2.2.5 [c.conv.5] - No desplazar operandos un número inválido de bits

ID	c.conv.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los operadores de desplazamiento de bits pueden generar **resultados impredecibles** cuando se utilizan con valores que exceden el tamaño del tipo de dato o con números negativos. Desplazar más bits que el ancho del operando izquierdo, o usar valores negativos como operando derecho, produce resultados impredecibles que varían entre compiladores y arquitecturas.

Codificación insegura

Se muestra cómo realizar desplazamientos de bits a partir de un valor proporcionado por el usuario, con el objetivo de entender qué ocurre cuando no se validan los límites del operando de desplazamiento y cómo esto puede provocar comportamiento indefinido.

```
#include <stdio.h>
#include <limits.h>

int main(void) {
  //-- Lee el número de bits a desplazar desde la entrada estándar
  int n;
  printf("Ingrese el numero de bits a desplazar: ");
  scanf("%d", &n);

  //-- Valor base para el desplazamiento
  unsigned int value = 1U;

  //-- Realiza el desplazamiento sin validar el rango de 'n'
  unsigned int result = value << n;

  //-- Muestra el resultado del desplazamiento
  printf("Resultado: %u\n", result);

  return 0;
}
```

Codificación recomendada

Se muestra cómo realizar desplazamientos de bits validando previamente que el número de bits esté dentro del rango permitido para el tipo de dato, evitando comportamiento indefinido cuando el desplazamiento es negativo o mayor o igual al tamaño en bits del operando.

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
  //-- Lee el número de bits a desplazar desde la entrada estándar
  int n;
  printf("Ingrese el numero de bits a desplazar: ");
  if (scanf("%d", &n) != 1) {
    //-- Maneja error de lectura
    printf("Entrada invalida.\n");
    return 1;
  }
  //-- Calcula el ancho en bits del tipo unsigned int
  unsigned int bit_width = sizeof(unsigned int) * CHAR_BIT;

  //-- Valida que el desplazamiento esté dentro del rango permitido
  if (n < 0 || (unsigned int)n >= bit_width) {
    //-- Rechaza valores que producen comportamiento indefinido
    printf("Numero de bits fuera de rango (0 a %u).\n", bit_width - 1);
    return 1;
  }
  //-- Valor base para el desplazamiento
  unsigned int value = 1U;
  unsigned int result = value << n;

  //-- Muestra el resultado seguro del desplazamiento
  printf("Resultado: %u\n", result);

  return 0;
}
```

2.2.5.1 Descripción

Esta regla establece que las operaciones de desplazamiento de bits deben realizarse únicamente con valores de desplazamiento válidos según el tipo de dato del operando izquierdo. El número de posiciones a desplazar debe ser no negativo y menor que el ancho en bits del tipo de dato que se está desplazando.

Los operadores de desplazamiento `<<` y `>>` requieren que el operando derecho tenga un valor dentro del rango permitido. Para un tipo de dato de n bits, los valores válidos de desplazamiento van desde 0 hasta n-1. Desplazar por un número de bits igual o mayor al ancho del tipo, o usar valores negativos, produce **comportamiento indefinido**.

El comportamiento indefinido en operaciones de desplazamiento puede manifestarse de diferentes formas según la arquitectura del procesador y el compilador utilizado. Algunos sistemas pueden producir resultados impredecibles, mientras que otros pueden generar valores que aparentan ser correctos, pero que en realidad son inconsistentes entre diferentes plataformas o configuraciones de compilación.

Los riesgos asociados incluyen la generación de valores incorrectos que pueden propagarse a través del programa, causando fallos lógicos difíciles de detectar. Estos errores pueden comprometer la integridad de cálculos críticos y crear vulnerabilidades de seguridad cuando se utilizan en operaciones de validación o control de acceso.

La mitigación efectiva requiere validar los valores de desplazamiento antes de realizar la operación. Esta validación debe verificar que el valor sea no negativo y esté dentro del rango permitido para el tipo de dato específico. El uso de constantes simbólicas para definir los límites máximos de desplazamiento mejora la legibilidad y mantenibilidad del código de validación.

Riesgos asociados

- Comportamiento indefinido del programa, lo que puede causar fallos inesperados.
- Pérdida o corrupción de datos debido a resultados aritméticos incorrectos.
- Vulnerabilidades de seguridad, incluyendo posibles desbordamientos y exposición de información sensible.
- Incompatibilidad entre diferentes compiladores o arquitecturas, dificultando la portabilidad del código.
- Fallos de lógica difíciles de detectar y depurar.
- Posible ejecución de código malicioso si el atacante logra manipular operaciones de desplazamiento.
- Producción de resultados inconsistentes que pueden afectar la integridad de procesos críticos.

Buenas prácticas

- Validar que el número de bits a desplazar sea mayor o igual a cero y menor que el tamaño en bits del operando.

```
//-- Desplazamiento sin validar el rango del contador de bits
unsigned int x = 1;
int n = user_input;
 n;
```

```
//-- Validación explícita del rango antes de desplazar
unsigned int x = 1;
int n = user_input;
if (n >= 0 && n < (int)(sizeof(x) * 8)) {
  unsigned int r = x << n;
}
```

- Utilizar tipos sin signo para operaciones de desplazamiento a nivel de bits.

```
//-- Desplazamiento sobre tipo con signo
int value = -1;
int r = value >> 1;
```

```
//-- Uso de tipo sin signo para evitar comportamiento indefinido
unsigned int value = 0xFFFFFFFFu;
unsigned int r = value >> 1;
```

- Evitar desplazamientos con valores negativos controlando explícitamente la entrada.

```
//-- Posible desplazamiento con valor negativo
int n = get_offset();
unsigned int r = 8u << n;
```

```
//-- Comprobación de que el desplazamiento no sea negativo
int n = get_offset();
if (n >= 0) {
  unsigned int r = 8u << n;
}
```

- Definir constantes o macros que representen el ancho en bits del tipo para evitar errores mágicos.

```
//-- Uso de valor literal que puede no coincidir con el tamaño real
unsigned int x = 1;
if (n < 32) {
  x <<= n;
}
```

```
//-- Cálculo del ancho en bits basado en sizeof
unsigned int x = 1;
#define UINT_BITS (sizeof(unsigned int) * 8)
if (n < UINT_BITS) {
  x <<= n;
}
```

- Evitar desplazar valores que puedan producir desbordamiento del bit más significativo.

```
//-- Desplazamiento que puede perder bits significativos
unsigned int x = 0x80000000u;
unsigned int r = x << 1;
```

```
//-- Verificación previa para evitar pérdida de bits
unsigned int x = 0x80000000u;
if ((x & (1u << ((sizeof(x) * 8) - 1))) == 0) {
  unsigned int r = x << 1;
}
```

2.2.5.2 Estándares metodológicos

- **CERT/C-INT34**: Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

2.2.6 [c.conv.6] - No convertir punteros a tipos con alineación más estricta

ID	c.conv.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los punteros pueden convertirse entre diferentes tipos, pero esta conversión genera problemas cuando el tipo destino requiere una **alineación más estricta** que el tipo origen. Esta situación podría causar fallos de ejecución, accesos incorrectos a memoria o corrupción de datos en arquitecturas que imponen requisitos específicos de alineación.

Codificación insegura

Uso de conversión de punteros desde un buffer de bytes hacia un tipo con requisitos de alineación más estrictos. Se busca mostrar cómo una conversión directa puede provocar comportamiento indefinido en arquitecturas donde la alineación es obligatoria, y entender el riesgo asociado.

```
#include <stdio.h>
#include <stdint.h>

int main(void) {
  //-- Se crea un buffer de bytes sin garantizar alineación para enteros
  unsigned char buffer[sizeof(int) + 1];

  //-- Se obtiene una dirección potencialmente desalineada
  unsigned char *ptr = buffer + 1;

  //-- Conversión directa a puntero a int (puede violar requisitos de alineación)
  int *int_ptr = (int *)ptr;

  //-- Escritura a través del puntero convertido
  *int_ptr = 12345;
```

```
  //-- Lectura del valor escrito
  printf("Valor: %d\n", *int_ptr);

  return 0;
}
```

Codificación recomendada

Uso seguro de un buffer de bytes para almacenar un valor entero sin convertir el puntero a un tipo con alineación más estricta. Se emplea copia de memoria para evitar accesos desalineados y mantener un comportamiento definido y portable.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main(void) {
  //-- Se crea un buffer de bytes
  unsigned char buffer[sizeof(int) + 1];

  //-- Se obtiene una dirección potencialmente desalineada
  unsigned char *ptr = buffer + 1;

  //-- Valor entero a almacenar
  int value = 12345;

  //-- Copia segura del entero al buffer sin convertir punteros
  memcpy(ptr, &value, sizeof(int));

  //-- Variable auxiliar correctamente alineada
  int read_value = 0;

  //-- Copia segura desde el buffer hacia variable alineada
  memcpy(&read_value, ptr, sizeof(int));

  //-- Uso del valor leído
  printf("Valor: %d\n", read_value);

  return 0;
}
```

2.2.6.1 Descripción

Esta regla prohíbe la conversión de punteros hacia tipos que requieren una alineación de memoria más estricta que el tipo original. La alineación de memoria se refiere a los requisitos específicos que tienen los diferentes tipos de datos sobre las direcciones de memoria donde pueden almacenarse de forma válida.

Cada tipo de dato tiene requisitos de alineación particulares. Los tipos básicos como `char` pueden almacenarse en cualquier dirección de memoria, mientras que tipos como `int`, `long` o `double` requieren que sus direcciones sean múltiplos de su tamaño natural. Cuando se convierte un puntero de un tipo con alineación menos restrictiva a otro con alineación más estricta, no existe garantía de que la dirección resultante cumpla con los requisitos del tipo destino.

El acceso a datos a través de punteros mal alineados constituye **comportamiento indefinido** según el estándar del lenguaje. En arquitecturas que requieren alineación estricta, este comportamiento puede provocar fallos de segmentación o excepciones de hardware que terminan abruptamente la ejecución del programa. En sistemas más permisivos, el acceso puede completarse, pero con penalizaciones significativas de rendimiento.

Los compiladores modernos pueden generar código optimizado asumiendo que los punteros están correctamente alineados. Cuando esta suposición es incorrecta, las optimizaciones pueden producir resultados inesperados o corrupción de datos, creando vulnerabilidades de seguridad difíciles de detectar y reproducir.

La mitigación de estos problemas requiere verificar que las conversiones de punteros mantengan la compatibilidad de alineación entre los tipos involucrados. Las herramientas de análisis estático pueden detectar muchas de estas conversiones problemáticas durante la compilación. En casos donde la conversión es necesaria, deben emplearse mecanismos que garanticen la alineación correcta antes de realizar el acceso a los datos.

Riesgos asociados

- Corrupción de memoria por accesos incorrectos a direcciones mal alineadas.
- Errores de ejecución (crash) debido a fallos de segmentación (segmentation fault) en arquitecturas que requieren alineación estricta.
- Comportamiento indefinido del programa según el estándar del lenguaje C.
- Vulnerabilidades de seguridad aprovechables por atacantes para corromper estructura de datos.
- Dificultad en la portabilidad del código a diferentes plataformas y arquitecturas.
- Pérdida de datos debido a lecturas/escrituras incorrectas en memoria.
- Reducción en el rendimiento, ya que algunos procesadores penalizan el acceso a datos mal alineados.
- Errores difíciles de depurar, ya que pueden depender del sistema operativo o hardware.

Buenas prácticas

- Evitar convertir punteros genéricos (`void*`) a tipos con requisitos de alineación más estrictos sin validar previamente la alineación.

```
//-- Conversión directa sin validar alineación
void procesar(void *buffer) {
  int *p = (int *)buffer;
  *p = 10;
}
```

```
//-- Verificación de alineación antes de convertir
void procesar(void *buffer) {
  if (((uintptr_t)buffer % alignof(int)) == 0) {
    int *p = (int *)buffer;
    *p = 10;
  }
}
```

- Utilizar `memcpy` para acceder a datos potencialmente desalineados en lugar de forzar conversiones de punteros.

```
//-- Conversión forzada de char* a double*
void leer_valor(char *data) {
  double *d = (double *)data;
  printf("%f", *d);
}
```

```
//-- Copia segura a variable local alineada
void leer_valor(char *data) {
  double valor;
  memcpy(&valor, data, sizeof(double));
  printf("%f", valor);
}
```

- Diseñar estructuras y buffers respetando la alineación natural del tipo más restrictivo que contendrán.

```
//-- Buffer de bytes reutilizado como estructura alineada
char buffer[sizeof(long)];
long *ptr = (long *)buffer;
*ptr = 12345;
```

```
//-- Uso de unión para garantizar alineación adecuada
union {
  char buffer[sizeof(long)];
```

```
  long align;
} datos;
long *ptr = (long *)datos.buffer;
*ptr = 12345;
```

2.2.6.2 Estándares metodológicos

- **CERT/C-EXP36**: Do not cast pointers into more strictly aligned pointer types.

2.3 Inicialización

2.3.1 [c.ini.1] - Inicializar todas las variables automáticas antes de su uso

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

Las variables automáticas no se inicializan automáticamente y contienen valores impredecibles al momento de su declaración. El uso de variables no inicializadas podría provocar fallos de seguridad, corrupción de datos o ejecución impredecible del programa. Esta vulnerabilidad es especialmente crítica cuando las variables contienen información sensible o controlan el flujo de ejecución.

Codificación insegura

Se muestra un programa que utiliza una variable automática local para realizar un cálculo y mostrar su valor por pantalla. La intención es observar qué ocurre cuando una variable local no se inicializa antes de ser utilizada y cómo esto afecta al comportamiento del programa.

```
#include <stdio.h>

int main(void) {
  //-- Declaración de una variable automática sin inicializar
  int contador;

  //-- Uso de la variable antes de asignarle un valor definido
  printf("Valor del contador: %d\n", contador);

  //-- Incremento basado en un valor indeterminado
  contador++;
  printf("Valor del contador tras incremento: %d\n", contador);

  return 0;
}
```

Codificación recomendada

Se muestra un programa similar donde la variable automática local se inicializa explícitamente antes de cualquier uso. El objetivo es garantizar un estado conocido desde el inicio, evitando comportamientos indeterminados y asegurando resultados predecibles.

```
#include <stdio.h>

int main(void) {
  //-- Declaración e inicialización de la variable automática
  int contador = 0;

  //-- Uso seguro de la variable con un valor definido
  printf("Valor inicial del contador: %d\n", contador);

  //-- Incremento basado en un valor conocido
  contador++;
  printf("Valor del contador tras incremento: %d\n", contador);

  return 0;
}
```

2.3.1.1 Descripción

Las variables automáticas no reciben inicialización implícita y contienen valores indeterminados en el momento de su declaración. El acceso a estas variables antes de asignarles un valor explícito constituye comportamiento indefinido según el estándar del lenguaje.

Esta situación genera **vulnerabilidades de seguridad** porque los valores residuales en memoria pueden contener información sensible de ejecuciones anteriores o datos de otros procesos. Los atacantes pueden explotar estas filtraciones para obtener claves criptográficas, contraseñas o información confidencial que permanece en posiciones de memoria no inicializadas.

El problema afecta especialmente a variables de tipo `char`, `int`, punteros y estructuras declaradas en el ámbito de función. Los compiladores modernos pueden detectar algunos casos mediante análisis estático, pero no garantizan la detección completa de todos los escenarios problemáticos.

La **mitigación efectiva** requiere establecer valores iniciales explícitos para todas las variables automáticas en el punto de declaración o mediante asignación antes del primer uso. Las herramientas de análisis estático y las opciones de compilación con advertencias habilitadas ayudan a identificar variables potencialmente no inicializadas durante el desarrollo.

Los entornos de ejecución seguros pueden implementar inicialización automática de memoria, pero esta protección no debe considerarse una solución definitiva debido a las variaciones entre implementaciones y el impacto en rendimiento que puede generar.

Riesgos asociados

- Comportamiento indefinido del programa debido al uso de valores no inicializados.
- Vulnerabilidades de seguridad como divulgación de información sensible almacenada en memoria residual.
- Errores lógicos difíciles de detectar y depurar.
- Resultados incorrectos en cálculos o toma de decisiones dentro del programa.
- Posibles fallos o interrupciones inesperadas (crash) del programa.
- Fallos de validación de argumentos o condiciones, abriendo la puerta a exploits.
- Difícil reproducibilidad y testeo de los errores en diferentes plataformas o compiladores.
- Aumento del ataque de inyección de datos a través de memoria reutilizada.

Buenas prácticas

- Inicializar variables automáticas en el momento de su declaración para evitar valores indeterminados.

```
//-- Uso de variable automática sin inicializar
int contador;
printf("%d", contador);
```

```
//-- Inicialización explícita en la declaración
int contador = 0;
printf("%d", contador);
```

- Asignar un valor inicial antes de utilizar variables en estructuras condicionales.

```
//-- Variable usada en condición sin valor definido
int bandera;
if (bandera) {
  procesar();
}
```

```
//-- Inicialización previa antes de evaluar la condición
int bandera = 0;
if (bandera) {
```

```
  procesar();
}
```

- Inicializar acumuladores y variables de cálculo antes de operaciones aritméticas.

```
//-- Acumulador sin inicializar antes de sumar
int suma;
suma += valor;
```

```
//-- Inicialización explícita del acumulador
int suma = 0;
suma += valor;
```

- Inicializar punteros automáticos antes de su uso para evitar referencias indeterminadas.

```
//-- Puntero automático sin inicializar
int *ptr;
*ptr = 10;
```

```
//-- Inicialización del puntero antes de dereferenciar
int valor = 0;
int *ptr = &valor;
*ptr = 10;
```

2.3.1.2 Estándares metodológicos

- **MISRA C-9.1**: All automatic variables shall have been assigned a value before being used.

2.3.2 [c.ini.2] - Declarar objetos con duraciones de almacenamiento apropiadas

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

Los objetos en memoria poseen diferentes **duraciones de almacenamiento** que determinan cuándo se crean, cuánto tiempo permanecen válidos y cuándo se destruyen. Una declaración incorrecta puede provocar **accesos a memoria no válida**, referencias a objetos ya destruidos o comportamientos indefinidos que comprometen la estabilidad del programa.

Codificación insegura

Gestión de la duración de almacenamiento de variables dentro de funciones. Se muestra cómo el uso inadecuado de almacenamiento estático puede provocar comportamientos no deseados al compartir estado entre llamadas, y cómo definir correctamente la duración automática evita efectos secundarios y mejora la seguridad y previsibilidad del programa.

```
#include <stdio.h>

//-- Función que usa una variable estática compartida entre llamadas
int incrementarContador() {
  //-- Variable estática: mantiene su valor entre llamadas
  static int contador = 0;
  contador++;
  return contador;
}
int main() {
  //-- Se realizan múltiples llamadas esperando un valor independiente
  printf("Primera llamada: %d\n", incrementarContador());
  printf("Segunda llamada: %d\n", incrementarContador());
  printf("Tercera llamada: %d\n", incrementarContador());

  return 0;
}
```

Codificación recomendada

Uso de variables con duración automática para evitar estado compartido no intencionado. Se define el objeto dentro del alcance adecuado y se pasa como argumento cuando es necesario, garantizando que cada ejecución tenga su propio estado y evitando efectos colaterales.

```
#include <stdio.h>

//-- Función que recibe el contador por parámetro
int incrementarContador(int contador) {
  //-- Variable automática: solo existe durante la ejecución de la función
  contador++;
  return contador;
}
int main() {
  //-- Variable automática en main con alcance controlado
  int contador = 0;

  contador = incrementarContador(contador);
  printf("Primera llamada: %d\n", contador);
```

```
    contador = incrementarContador(contador);
    printf("Segunda llamada: %d\n", contador);

    contador = incrementarContador(contador);
    printf("Tercera llamada: %d\n", contador);

    return 0;
}
```

2.3.2.1 Descripción

La duración de almacenamiento determina el tiempo de vida de los objetos durante la ejecución del programa. Una declaración inadecuada puede provocar accesos a memoria no válida, corrupción de datos o comportamiento indefinido del programa.

Los objetos pueden tener **duración automática**, **estática**, **asignada** o **de hilo**. Cada tipo presenta características específicas sobre cuándo se crean, inicializan y destruyen los objetos. La elección incorrecta de la duración puede generar problemas de acceso a memoria liberada, referencias a objetos inexistentes o conflictos de concurrencia.

Los objetos con duración automática se crean al entrar en su ámbito y se destruyen al salir de él. Su dirección de memoria no debe utilizarse fuera de su ámbito de vida. Los objetos con duración estática persisten durante toda la ejecución del programa, mientras que los de duración asignada requieren gestión manual de memoria.

La duración de hilo, disponible mediante `_Thread_local`, vincula la vida del objeto al hilo de ejecución correspondiente. Su uso inadecuado en entornos multihilo puede causar condiciones de carrera o accesos a memoria no válida.

Los riesgos asociados incluyen **use-after-free**, **dangling pointers** y **memory leaks**. Estos problemas pueden comprometer la estabilidad del programa y crear vulnerabilidades de seguridad explotables.

La mitigación requiere análisis cuidadoso del ciclo de vida de cada objeto y selección de la duración apropiada según su uso previsto. Las herramientas de análisis estático pueden detectar algunos casos problemáticos, aunque no todos los escenarios son verificables automáticamente.

Riesgos asociados

- Lectura de memoria no inicializada (uso de variables no inicializadas).
- Condiciones de carrera y problemas de concurrencia por objetos con almacenamiento compartido inadecuado.
- Desbordamientos de buffer debido a un tiempo de vida insuficiente del objeto.
- Acceso a memoria liberada (dangling pointers o uso después de liberar).

- Fugas de memoria por objetos que exceden su duración necesaria.
- Comportamiento indefinido por uso de variables automáticas fuera de su alcance.
- Violaciones de integridad de datos por objetos globales o estáticos mal usados.
- Dificultad para el mantenimiento y la auditoría del código debido a la confusión en la duración y visibilidad de los objetos.
- Posibilidad de exposición de información sensible en memoria tras finalizar el ciclo de vida de un objeto.

Buenas prácticas

- Evitar devolver direcciones de variables con duración automática (stack) y utilizar almacenamiento estático o dinámico cuando el objeto deba sobrevivir al ámbito.

```
//-- Retorno de dirección de variable automática que deja de existir fuera del
ámbito
char *obtenerMensaje() {
  char buffer[32] = "Hola";
  return buffer;
}
```

```
//-- Uso de almacenamiento estático para mantener validez tras la salida de la
función
char *obtenerMensaje() {
  static char buffer[32] = "Hola";
  return buffer;
}
```

- Evitar el uso innecesario de variables globales y preferir variables locales o pasar parámetros explícitamente.

```
//-- Uso de variable global con exposición innecesaria
int contador;
void incrementar() {
  contador++;
}
```

```
//-- Variable local pasada como parámetro para limitar el alcance
void incrementar(int *contador) {
  (*contador)++;
}
```

- Utilizar almacenamiento dinámico cuando el tamaño del objeto dependa de datos en tiempo de ejecución y validar su liberación.

```
//-- Array automático con tamaño fijo que puede ser insuficiente
void procesar(int n) {
  int datos[10];
  /* uso de datos con n potencialmente mayor que 10 */
}
```

```
//-- Reserva dinámica ajustada al tamaño requerido en ejecución
void procesar(int n) {
  int *datos = malloc(n * sizeof(int));
  if (datos == NULL) return;
  /* uso de datos */
  free(datos);
}
```

- Evitar el uso de variables automáticas no inicializadas y asegurar su correcta inicialización dentro del ámbito adecuado.

```
//-- Variable automática sin inicializar utilizada antes de asignación
void funcion() {
  int bandera;
  if (bandera) {
    /* ... */
  }
}
```

```
//-- Inicialización explícita dentro del ámbito correcto
void funcion() {
  int bandera = 0;
  if (bandera) {
    /* ... */
  }
}
```

- Limitar la duración y visibilidad de objetos estáticos utilizando el modificador `static` a nivel de archivo cuando no deban exponerse externamente.

```
//-- Variable global visible desde otros archivos
int claveSecreta = 1234;
```

```
//-- Variable con duración estática y alcance limitado al archivo
static int claveSecreta = 1234;
```

2.3.2.2 Estándares metodológicos

- **CERT/C-DCL30**: Declare objects with appropriate storage durations.

2.3.3 [c.ini.3] - Usar la sintaxis correcta para miembros de arrays flexibles

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

Los **arrays flexibles** permiten definir estructuras con un último miembro de tamaño variable, pero su declaración incorrecta puede generar comportamientos indefinidos y problemas de compatibilidad. La sintaxis inadecuada en la declaración del miembro flexible o el uso incorrecto de `sizeof` con estas estructuras compromete la **portabilidad del código** y puede provocar errores de acceso a memoria en tiempo de ejecución.

Codificación insegura

Gestión de una estructura que almacena una longitud y un bloque de datos variable en memoria dinámica. Se muestra cómo se reserva memoria y se accede al array interno, destacando qué ocurre cuando no se usa correctamente la sintaxis de miembros de array flexibles y se realizan cálculos de tamaño incorrectos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Buffer {
  //-- Estructura con un array de tamaño fijo 1 usado como si fuera flexible
  size_t len;
  char data[1];
};

int main(void) {
  //-- Longitud deseada para los datos
  size_t length = 10;

  //-- Reserva incorrecta: no se ajusta el tamaño restando el byte ya incluido en
data[1]
  struct Buffer *buf = malloc(sizeof(struct Buffer) + length);
  if (buf == NULL) {
    perror("malloc");
    return 1;
  }
  //-- Inicialización de la estructura
  buf->len = length;
  memset(buf->data, 'A', length);

  //-- Impresión del contenido
  for (size_t i = 0; i < buf->len; i++) {
    putchar(buf->data[i]);
  }
```

```
  putchar('\n');

  //-- Liberación de memoria
  free(buf);
  return 0;
}
```

Codificación recomendada

Gestión segura de una estructura con un miembro de array flexible declarado correctamente según el estándar C (**C99 o superior**). Se muestra cómo definir el array sin tamaño explícito y cómo calcular el tamaño total de memoria dinámicamente para evitar errores de desbordamiento y mantener coherencia con el estándar.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Buffer {
  //-- Estructura con miembro de array flexible declarado correctamente
  size_t len;
  char data[];
};

int main(void) {
  //-- Longitud deseada para los datos
  size_t length = 10;

  //-- Reserva correcta usando sizeof(struct) + longitud necesaria
  struct Buffer *buf = malloc(sizeof(struct Buffer) + length);
  if (buf == NULL) {
    perror("malloc");
    return 1;
  }
  //-- Inicialización de la estructura
  buf->len = length;
  memset(buf->data, 'A', length);

  //-- Impresión del contenido
  for (size_t i = 0; i < buf->len; i++) {
    putchar(buf->data[i]);
  }
  putchar('\n');

  //-- Liberación de memoria
  free(buf);
  return 0;
```

```
}
```

2.3.3.1 Descripción

La característica de los **arrays flexibles**, introducida en C99, requiere una sintaxis específica donde el último miembro se declara como un array sin especificar su tamaño, utilizando corchetes vacíos.

La declaración incorrecta de estos datos puede generar **comportamiento indefinido** durante la compilación y ejecución. Los compiladores pueden interpretar de manera inconsistente las declaraciones malformadas, lo que resulta en asignaciones de memoria incorrectas o accesos fuera de los límites esperados.

El estándar establece restricciones específicas sobre la ubicación y declaración de estos miembros. Solo pueden aparecer como el **último elemento** de una estructura, y la estructura debe contener al menos un miembro adicional antes del array flexible. Además, no pueden utilizarse en arrays de estructuras ni como miembros de uniones.

Los riesgos asociados incluyen **corrupción de memoria**, accesos no autorizados a regiones de memoria adyacentes y fallos de segmentación. Estos problemas son especialmente críticos en aplicaciones que manejan datos sensibles u operan en entornos con restricciones de memoria.

La mitigación efectiva requiere seguir estrictamente la sintaxis estándar y verificar que las herramientas de compilación soporten completamente C99 o versiones posteriores. Es recomendable utilizar herramientas de análisis estático que detecten declaraciones incorrectas y validar el comportamiento mediante pruebas exhaustivas de asignación y acceso a memoria.

Riesgos asociados

- Desbordamientos de buffer por acceso fuera de los límites del array flexible.
- Corrupción de memoria debido a escritura en posiciones no reservadas adecuadamente.
- Lectura de información errónea o sensible de la memoria adyacente al array.
- Comportamiento indefinido del programa, pudiendo resultar en fallos o bloqueos.
- Vulnerabilidad a la explotación por atacantes que busquen ejecutar código malicioso.
- Dificultad para mantener y depurar el código debido a inconsistencias en la estructura de datos.
- Problemas de portabilidad entre compiladores o plataformas que manejan diferente la sintaxis incorrecta.

- Fallos en la inicialización y el cálculo del tamaño correcto de las estructuras que incluyen arrays flexibles.

Buenas prácticas

- Declarar el array flexible como último miembro de la estructura y sin tamaño explícito.

```
//-- Declaración con tamaño explícito que rompe la semántica de array flexible
struct paquete {
  size_t longitud;
  char datos[1];
};
```

```
//-- Declaración correcta del miembro flexible como último campo sin tamaño
struct paquete {
  size_t longitud;
  char datos[];
};
```

- Reservar memoria considerando `sizeof(struct)` más el tamaño real requerido para el array flexible.

```
//-- Reserva usando solo sizeof sin espacio adicional para los datos variables
size_t n = 100;
struct paquete *p = malloc(sizeof(struct paquete));
```

```
//-- Reserva sumando el tamaño base y el espacio necesario para el array flexible
size_t n = 100;
struct paquete *p = malloc(sizeof(struct paquete) + n * sizeof(char));
```

- No usar `sizeof` sobre el miembro flexible para calcular su longitud real.

```
//-- Uso de sizeof sobre el array flexible que no refleja la memoria asignada
dinámicamente
size_t tam = sizeof(p->datos);
```

```
//-- Uso de un campo explícito que almacena la longitud válida del array flexible
size_t tam = p->longitud;
```

- Evitar copiar datos al array flexible sin validar previamente el tamaño reservado.

```
//-- Copia sin comprobar que el tamaño reservado es suficiente
memcpy(p->datos, origen, longitud_origen);
```

```
//-- Validación previa usando el campo de longitud antes de copiar
if (longitud_origen <= p->longitud) {
  memcpy(p->datos, origen, longitud_origen);
}
```

2.3.3.2 Estándares metodológicos

- **CERT/C-DCL38**: Use the correct syntax when declaring a flexible array member.

2.3.4 [c.ini.4] - No filtrar información al pasar estructuras a través de límites de confianza

ID	c.ini.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las estructuras pueden contener **padding bytes** no inicializados entre sus miembros debido a los requisitos de alineación del compilador. Cuando estas estructuras se transmiten a través de **límites de confianza** como interfaces de red, archivos o comunicación entre procesos, los datos residuales en el padding pueden revelar información sensible de la memoria.

Codificación insegura

Se muestra el envío de una estructura que contiene información sensible a través de un límite de confianza, como podría ser una función que simula enviar datos a un sistema externo. El objetivo es entender cómo los campos internos no destinados a compartirse pueden filtrarse accidentalmente si no se controla explícitamente qué información se expone.

```
#include <stdio.h>
#include <string.h>

//-- Estructura interna con datos sensibles
typedef struct {
  char username[50];
  char password[50];
  char internal_token[50];
} User;

//-- Simula el envío de datos a un sistema externo (límite de confianza)
void sendToExternalSystem(const void *data, size_t size) {
  //-- En un caso real, aquí se enviaría por red o a otro proceso
  fwrite(data, 1, size, stdout);
}
int main() {
```

```
  //-- Inicialización de la estructura con datos sensibles
  User user;
  strcpy(user.username, "alice");
  strcpy(user.password, "SuperSecret123");
  strcpy(user.internal_token, "TOKEN-ABC-123");

  //-- Se envía toda la estructura sin filtrar campos sensibles
  sendToExternalSystem(&user, sizeof(User));

  return 0;
}
```

Codificación recomendada

Se presenta el control explícito de la información que cruza un límite de confianza, creando una estructura específica para la transferencia de datos. Se busca demostrar cómo limitar los campos expuestos evita la filtración de información sensible contenida en estructuras internas.

```
#include <stdio.h>
#include <string.h>

//-- Estructura interna con datos sensibles
typedef struct {
  char username[50];
  char password[50];
  char internal_token[50];
} User;

//-- Estructura segura para transferencia externa
typedef struct {
  char username[50];
} PublicUserData;

//-- Simula el envío de datos a un sistema externo (límite de confianza)
void sendToExternalSystem(const void *data, size_t size) {
  //-- En un caso real, aquí se enviaría por red o a otro proceso
  fwrite(data, 1, size, stdout);
}
int main() {
  //-- Inicialización de la estructura interna
  User user;
  strcpy(user.username, "alice");
  strcpy(user.password, "SuperSecret123");
  strcpy(user.internal_token, "TOKEN-ABC-123");

  //-- Preparación de estructura filtrada para el exterior
  PublicUserData publicData;
```

```
  memset(&publicData, 0, sizeof(PublicUserData));
  strncpy(publicData.username, user.username, sizeof(publicData.username) - 1);

  //-- Solo se envían los datos necesarios
  sendToExternalSystem(&publicData, sizeof(PublicUserData));

  return 0;
}
```

2.3.4.1 Descripción

Las estructuras de datos pueden contener información residual en espacios no utilizados debido al relleno automático que aplican los compiladores para optimizar el alineamiento en memoria. Este relleno, conocido como padding, introduce bytes cuyo contenido no está definido por el programador y puede contener datos sensibles de operaciones previas.

Cuando estas estructuras se transmiten a través de límites de confianza, como interfaces de red, comunicación entre procesos o almacenamiento persistente, la información residual puede quedar expuesta inadvertidamente. Los atacantes pueden explotar esta filtración para obtener datos confidenciales que no estaban destinados a ser compartidos.

El problema se agrava porque el comportamiento del relleno depende de la arquitectura del sistema, el compilador utilizado y las opciones de compilación. Esta variabilidad hace que el código aparentemente seguro en un entorno pueda presentar vulnerabilidades en otro.

Los riesgos asociados incluyen la **exposición de información confidencial**, como contraseñas, claves criptográficas o datos personales que pueden residir en la memoria no inicializada. Esta filtración puede comprometer la confidencialidad del sistema y facilitar ataques posteriores.

La mitigación efectiva requiere garantizar que toda la estructura esté completamente inicializada antes de su transmisión. Esto incluye tanto los campos definidos por el programador como los espacios de relleno insertados automáticamente. Las técnicas de inicialización deben ser explícitas y verificables, asegurando que no quede información residual en ninguna parte de la estructura.

Riesgos asociados

- Filtración de información sensible o confidencial a partes no autorizadas.
- Divulgación accidental de datos no inicializados dentro de la estructura.
- Exposición de datos sensibles del sistema o de la aplicación, lo que puede facilitar ataques de ingeniería inversa.

- Incremento de la superficie de ataque debido a la revelación de punteros, identificadores o direcciones de memoria.
- Facilitación de ataques de escalada de privilegios si se filtran tokens de autenticación o credenciales.
- Aumento del riesgo de exploits de tipo memory disclosure que podrían ser utilizados junto a vulnerabilidades como buffer overflow.
- Quebranto del cumplimiento normativo o de regulaciones sobre manejo de datos personales (por ejemplo, GDPR, HIPAA).
- Riesgo reputacional y pérdida de confianza de usuarios o clientes.

Buenas prácticas

- Definir estructuras específicas para intercambio externo que excluyan campos sensibles o internos antes de cruzar límites de confianza.

```
//-- Se envía la estructura completa con campos internos
typedef struct {
  int id;
  char nombre[50];
  char password_hash[64];
} Usuario;
enviar(socket, &usuario, sizeof(Usuario), 0);
```

```
//-- Se define una estructura pública sin información sensible
typedef struct {
  int id;
  char nombre[50];
} UsuarioPublico;
UsuarioPublico pub = { usuario.id, "" };
strncpy(pub.nombre, usuario.nombre, sizeof(pub.nombre) - 1);
enviar(socket, &pub, sizeof(UsuarioPublico), 0);
```

- Serializar explícitamente los campos permitidos en lugar de copiar memoria completa de la estructura.

```
//-- Se transmite la memoria completa incluyendo padding y datos no inicializados
struct Config {
  int modo;
  char ruta[100];
};
struct Config cfg;
cfg.modo = 1;
strcpy(cfg.ruta, "/tmp/data");
send(fd, &cfg, sizeof(cfg), 0);
```

```
//-- Se serializan únicamente los campos necesarios de forma controlada
char buffer[128];
int offset = 0;
memcpy(buffer + offset, &cfg.modo, sizeof(cfg.modo));
offset += sizeof(cfg.modo);
memcpy(buffer + offset, cfg.ruta, strlen(cfg.ruta) + 1);
offset += strlen(cfg.ruta) + 1;
send(fd, buffer, offset, 0);
```

- Inicializar completamente las estructuras antes de su uso para evitar fugas de memoria residual.

```
//-- Se envía una estructura con campos no inicializados
struct Respuesta {
  int codigo;
  char mensaje[64];
};
struct Respuesta r;
r.codigo = 200;
strcpy(r.mensaje, "OK");
write(fd, &r, sizeof(r));
```

```
//-- Se limpia la estructura antes de asignar valores y transmitirla
struct Respuesta r2;
memset(&r2, 0, sizeof(r2));
r2.codigo = 200;
strncpy(r2.mensaje, "OK", sizeof(r2.mensaje) - 1);
write(fd, &r2, sizeof(r2));
```

2.3.4.2 Estándares metodológicos

- **CERT/C-DCL39**: Avoid information leakage when passing a structure across a trust boundary.

2.3.5 [c.ini.5] - No declarar variables dentro de un switch antes de la primera etiqueta case

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

La declaración de variables entre la instrucción `switch` y la primera etiqueta `case` genera **código inalcanzable** que nunca se ejecuta. Esta práctica produce comportamientos impredecibles y dificulta el mantenimiento del código, ya que las variables declaradas en esta zona quedan en un estado indefinido.

Codificación insegura

Uso de una estructura `switch` donde se declaran variables antes de la primera etiqueta `case`. Se busca entender cómo esta práctica puede generar problemas de compilación, advertencias o comportamientos no esperados debido al flujo de ejecución y al alcance de las variables dentro del bloque `switch`.

```
#include <stdio.h>

//-- Programa que muestra un uso incorrecto de variables dentro de un switch
int main() {
  int opcion = 1;

  switch (opcion) {
    //-- Declaración antes del primer case (mala práctica)
    int valor = 10;

    case 1:
      printf("Opcion 1, valor = %d\n", valor);
      break;

    case 2:
      printf("Opcion 2\n");
      break;

    default:
      printf("Opcion no valida\n");
      break;
  }
  return 0;
}
```

Codificación recomendada

Uso correcto de una estructura `switch` donde las variables se declaran dentro de cada bloque `case`, asegurando un alcance adecuado y evitando problemas de compilación o ambigüedad en el flujo de ejecución. Se refuerza la buena práctica de delimitar cada caso con llaves cuando se requieren variables locales.

```
#include <stdio.h>

//-- Programa que muestra un uso correcto de variables dentro de un switch
int main() {
  int opcion = 1;

  switch (opcion) {
    case 1: {
      //-- Declaración dentro del bloque del case
```

```
      int valor = 10;
      printf("Opcion 1, valor = %d\n", valor);
      break;
    }
    case 2: {
      printf("Opcion 2\n");
      break;
    }
    default: {
      printf("Opcion no valida\n");
      break;
    }
  }
  return 0;
}
```

2.3.5.1 Descripción

Esta regla establece que no se deben declarar variables en el ámbito de una sentencia `switch` antes de la primera etiqueta `case` o `default`. Las declaraciones colocadas inmediatamente después de la llave de apertura del `switch` y antes de cualquier etiqueta crean un comportamiento impredecible en el flujo de ejecución.

El problema surge porque el control de flujo en una estructura `switch` salta directamente a la etiqueta correspondiente, omitiendo cualquier código que se encuentre entre la apertura del bloque y la primera etiqueta. Esto significa que las declaraciones de variables en esta zona nunca se ejecutan, aunque las variables permanecen en el ámbito léxico del bloque completo.

Esta situación genera **variables no inicializadas** que técnicamente están en el ámbito, pero nunca han sido declaradas en tiempo de ejecución. El acceso a estas variables desde cualquier `case` produce comportamiento indefinido, ya que se intenta utilizar memoria que no ha sido correctamente asignada.

Los riesgos asociados incluyen corrupción de memoria, fallos de segmentación y vulnerabilidades de seguridad derivadas del uso de datos no inicializados. El comportamiento indefinido puede manifestarse de forma inconsistente entre diferentes compiladores, versiones o configuraciones de optimización.

La mitigación se logra reubicando las declaraciones dentro de bloques específicos de cada `case` cuando sea necesario, o declarando las variables fuera de la estructura `switch` antes de su uso. Alternativamente, se puede colocar toda la lógica del `switch` dentro de un bloque adicional que contenga las declaraciones necesarias.

Esta práctica mejora la **predictibilidad del código** y elimina ambigüedades sobre el ciclo de vida de las variables, facilitando tanto la detección de errores durante la compilación como el mantenimiento del código.

Riesgos asociados

- Comportamiento indefinido si la inicialización de la variable es omitida debido al flujo de ejecución del `switch`.
- Errores de compilación o warnings difíciles de entender y depurar.
- Acceso a variables no inicializadas si se salta la inicialización mediante case labels.
- Vulnerabilidades que pueden ser explotadas si la variable contiene datos residuales no previstos.
- Perdida de portabilidad del código, ya que el comportamiento puede variar entre compiladores.
- Incremento en la dificultad de mantenimiento y legibilidad del código.
- Mayor probabilidad de defectos lógicos en la aplicación.

Buenas prácticas

- Declarar las variables antes del `switch` para evitar problemas de alcance y saltos directos a etiquetas `case`.

```
//-- Declaración dentro del switch antes de case
switch(opcion) {
  int valor = 0;
  case 1:
    valor = 10;
    break;
}
```

```
//-- Declaración antes del switch
int valor = 0;
switch(opcion) {
  case 1:
    valor = 10;
    break;
}
```

- Encapsular en un bloque (`{ }`) el código específico de cada `case` cuando se necesiten variables locales.

```
//-- Variable declarada directamente tras case sin bloque
switch(tipo) {
  case 2:
    int contador = 5;
    procesar(contador);
    break;
```

```
}
```

```
//-- Uso de bloque para delimitar el alcance de la variable
switch(tipo) {
  case 2: {
    int contador = 5;
    procesar(contador);
    break;
  }
}
```

- Inicializar previamente las variables que serán utilizadas en múltiples `case` para evitar uso de valores indeterminados.

```
//-- Variable declarada en switch sin inicialización clara
switch(estado) {
  int resultado;
  case 0:
    resultado = calcular();
    break;
  case 1:
    usar(resultado);
    break;
}
```

```
//-- Variable declarada e inicializada antes del switch
int resultado = 0;
switch(estado) {
  case 0:
    resultado = calcular();
    break;
  case 1:
    usar(resultado);
    break;
}
```

- Mantener el `switch` libre de declaraciones antes del primer `case` para cumplir con estándares como MISRA C y evitar comportamientos indefinidos.

```
//-- Declaración antes del primer case dentro del switch
switch(comando) {
  char buffer[10];
  case 'A':
    procesarA();
    break;
}
```

```
//-- Declaraciones fuera del switch conforme a estándares
char buffer[10];
switch(comando) {
  case 'A':
    procesarA();
    break;
}
```

2.3.5.2 Estándares metodológicos

- **CERT/C-DCL41**: Do not declare variables inside a switch statement before the first case label.

2.3.6 [c.ini.6] - No acceder a objetos mediante punteros de tipo incompatible

ID	c.ini.6	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El acceso a objetos mediante punteros de tipo incompatible genera un **comportamiento impredecible** y puede provocar corrupción de datos, fallos de segmentación o vulnerabilidades de seguridad. Esta práctica viola las reglas de **strict aliasing** y puede resultar en interpretaciones incorrectas de los datos almacenados en memoria, especialmente cuando se realizan conversiones entre tipos de diferentes tamaños o representaciones.

Codificación insegura

Acceso a memoria interpretando un objeto a través de un puntero de tipo diferente, forzando un cast y leyendo el contenido como si fuera otro tipo. Se pretende mostrar cómo esta práctica puede producir comportamiento indefinido debido a violaciones de aliasing estricto y diferencias de representación en memoria.

```
#include <stdio.h>

//-- Estructura con dos campos enteros
struct Punto {
  int x;
  int y;
};

int main() {
  //-- Crear e inicializar una estructura
  struct Punto p;
  p.x = 10;
  p.y = 20;
```

```
  //-- Forzar la interpretación de la estructura como si fuera un double
  //-- Esto viola las reglas de aliasing y produce comportamiento indefinido
  double *ptr = (double *)&p;

  //-- Acceso a memoria con tipo incompatible
  printf("Valor interpretado como double: %f\n", *ptr);

  return 0;
}
```

Codificación recomendada

Acceso seguro a los datos respetando el tipo real del objeto y evitando conversiones de punteros incompatibles. Se pretende mostrar cómo trabajar con los miembros de la estructura directamente o mediante punteros del mismo tipo, garantizando comportamiento definido y portable.

```
#include <stdio.h>

//-- Estructura con dos campos enteros
struct Punto {
  int x;
  int y;
};

int main() {
  //-- Crear e inicializar una estructura
  struct Punto p;
  p.x = 10;
  p.y = 20;

  //-- Usar un puntero del mismo tipo que el objeto
  struct Punto *ptr = &p;

  //-- Acceso seguro a los campos mediante el tipo correcto
  printf("x: %d\n", ptr->x);
  printf("y: %d\n", ptr->y);

  return 0;
}
```

2.3.6.1 Descripción

Esta regla prohíbe acceder a objetos en memoria utilizando punteros que no corresponden al tipo real del objeto almacenado. El problema surge cuando se realiza

una conversión de puntero y posteriormente se desreferencia para leer o escribir datos, interpretando incorrectamente la representación en memoria del objeto original.

El estándar define el concepto de **tipo efectivo** de un objeto, que determina cómo debe interpretarse su representación en memoria. Cuando un puntero de un tipo se convierte a otro tipo incompatible y se utiliza para acceder al objeto, se viola esta regla fundamental del modelo de memoria.

Las violaciones más comunes ocurren al convertir punteros entre tipos de diferentes tamaños, alineaciones o interpretaciones de bits. También se presenta cuando se accede a estructuras o uniones mediante punteros a tipos que no coinciden con la definición original del objeto.

Los riesgos asociados incluyen **comportamiento indefinido** según el estándar, corrupción de datos debido a interpretaciones incorrectas de la representación en memoria, y fallos de ejecución por problemas de alineación. Los compiladores modernos pueden generar código optimizado que asume el cumplimiento de las reglas de aliasing, lo que puede producir resultados inesperados cuando se violan.

La mitigación se basa en mantener la **coherencia de tipos** durante todo el ciclo de vida de los objetos. Esto implica utilizar las conversiones de tipo apropiadas según el estándar, emplear uniones cuando sea necesario acceder a la misma memoria con diferentes interpretaciones, y recurrir a funciones como `memcpy` para transferencias de datos entre tipos incompatibles. El uso de herramientas de análisis estático y compiladores con advertencias habilitadas ayuda a detectar violaciones potenciales durante el desarrollo.

Riesgos asociados

- Lectura o escritura inesperada de datos debido a la interpretación incorrecta del formato de los datos en memoria.
- Accesos desalineados a memoria, que pueden ocasionar fallos en arquitecturas que no permiten dichos accesos.
- Comportamiento indefinido, lo que podría ser explotado para ejecutar código malicioso o evadir mecanismos de seguridad.
- Corrupción de datos, ya que se pueden sobrescribir valores inesperadamente.
- Vulnerabilidades de seguridad como desbordamientos de buffer o uso de memoria no inicializada.
- Dificultad en el mantenimiento y depuración del código debido a errores difíciles de rastrear.
- Incompatibilidad con futuras versiones del compilador u otras plataformas debido a la dependencia de comportamientos no estándar.
- Fugas de información sensible si los datos accedidos contienen información

privada que se interpreta de forma incorrecta.

Buenas prácticas

- Evitar conversiones de punteros entre tipos incompatibles; usar el tipo original o un tipo compatible para acceder al objeto.

```
//-- Conversión de puntero a tipo incompatible para acceder al objeto
int value = 10;
float *fp = (float *)&value;
*fp = 3.14f;
```

```
//-- Uso del mismo tipo de puntero que el objeto original
int value2 = 10;
int *ip = &value2;
*ip = 20;
```

- No reinterpretar estructuras mediante punteros a otros tipos; utilizar el tipo correcto o funciones de copia.

```
//-- Reinterpretación de estructura como tipo incompatible
struct A { int x; };
struct B { float y; };
struct A a = { 5 };
struct B *bp = (struct B *)&a;
bp->y = 1.5f;
```

```
//-- Acceso utilizando el tipo de estructura correspondiente
struct A a2 = { 5 };
struct A *ap = &a2;
ap->x = 10;
```

- Evitar acceder a un bloque de memoria como si fuera otro tipo distinto; usar `memcpy` para reinterpretar representaciones.

```
//-- Acceso directo a memoria con puntero incompatible
double d = 3.1416;
int *ip2 = (int *)&d;
int v = *ip2;
```

```
//-- Copia de representación usando memcpy
double d2 = 3.1416;
int v2;
memcpy(&v2, &d2, sizeof(int));
```

- No acceder a arrays mediante punteros de tipo diferente; mantener coherencia

entre el tipo del array y el puntero.

```
//-- Acceso a array de int mediante puntero a short
int arr[2] = { 1, 2 };
short *sp = (short *)arr;
short s = sp[1];
```

```
//-- Acceso usando puntero del mismo tipo que el array
int arr2[2] = { 1, 2 };
int *ip3 = arr2;
int s2 = ip3[1];
```

2.3.6.2 Estándares metodológicos

- **CERT/C-EXP39**: Do not access a variable through a pointer of an incompatible type.

2.3.7 [c.ini.7] - No acceder a objetos volatile a través de referencias no volatile

ID	c.ini.7	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los objetos declarados como `volatile` requieren acceso directo a través de referencias que mantengan esta cualificación. En el acceso mediante punteros o referencias no `volatile`, el compilador asume que estos objetos no cambian de forma impredecible y puede aplicar optimizaciones incorrectas que ignoren modificaciones externas al flujo normal del programa.

Codificación insegura

Acceso a una variable compartida declarada como `volatile` a través de un puntero sin el calificador `volatile`, lo que elimina las garantías de acceso previstas por el compilador. Se pretende mostrar cómo la pérdida del calificador puede provocar optimizaciones incorrectas y comportamientos indefinidos en presencia de señales o concurrencia.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

//-- Variable global modificada por una señal
volatile int flag = 0;

//-- Manejador de señal que modifica la variable volatile
void handler(int sig) {
```

```
  //-- Se establece la bandera al recibir la señal
  flag = 1;
}
int main() {
  //-- Registro del manejador de señal
  signal(SIGALRM, handler);

  //-- Programar una alarma en 1 segundo
  alarm(1);

  //-- Puntero sin el calificador volatile (pierde la semántica original)
  int *ptr = (int *)&flag;

  //-- Espera activa hasta que la bandera cambie
  while (*ptr == 0) {
    //-- Bucle vacío
  }
  printf("Señal recibida, flag = %d\n", *ptr);
  return 0;
}
```

Codificación recomendada

Acceso a una variable compartida declarada como `volatile` manteniendo el calificador en todas las referencias, incluyendo punteros. Se muestra cómo preservar la semántica correcta para evitar optimizaciones indebidas y garantizar lecturas reales en cada acceso.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

//-- Variable global modificada por una señal
volatile int flag = 0;

//-- Manejador de señal que modifica la variable volatile
void handler(int sig) {
  //-- Se establece la bandera al recibir la señal
  flag = 1;
}
int main() {
  //-- Registro del manejador de señal
  signal(SIGALRM, handler);

  //-- Programar una alarma en 1 segundo
  alarm(1);

  //-- Puntero que conserva el calificador volatile
```

```
  volatile int *ptr = &flag;

  //-- Espera activa hasta que la bandera cambie
  while (*ptr == 0) {
    //-- Bucle vacío
  }
  printf("Señal recibida, flag = %d\n", *ptr);
  return 0;
}
```

2.3.7.1 Descripción

El calificador `volatile` instruye al compilador para que trate el objeto como potencialmente modificable por factores externos al flujo normal del programa, como interrupciones de hardware, señales del sistema operativo o acceso concurrente desde otros hilos de ejecución. Esta designación impide optimizaciones del compilador que podrían eliminar o reordenar accesos aparentemente redundantes.

Cuando se accede a un objeto `volatile` mediante una referencia no `volatile`, el compilador pierde la información sobre la naturaleza especial del objeto. Esto permite aplicar optimizaciones que asumen que el valor no cambia entre accesos consecutivos, generando código que puede omitir lecturas o escrituras necesarias.

Los riesgos asociados incluyen **lecturas de valores obsoletos**, **pérdida de escrituras críticas** y **violación de protocolos de sincronización** con hardware o sistemas concurrentes. Estos problemas se manifiestan típicamente como comportamientos intermitentes difíciles de reproducir y depurar.

La mitigación se basa en **mantener la consistencia de calificadores** a lo largo de toda la cadena de acceso. Las conversiones de tipo deben preservar el calificador `volatile`, y las funciones que operan sobre datos `volatile` deben declarar sus parámetros con la cualificación apropiada. El diseño de interfaces debe considerar explícitamente qué datos requieren semántica `volatile` y propagar esta información a través de las declaraciones de tipos y funciones.

Riesgos asociados

- Lectura o escritura de valores inconsistentes en tiempo de ejecución debido a optimizaciones del compilador.
- Violación de la coherencia de datos en entornos multihilo, llevando a condiciones de carrera.
- Comportamiento indefinido del programa por acceso no sincronizado a memoria compartida.
- Dificultad para identificar y depurar errores que se manifiestan solo bajo ciertas condiciones de ejecución.

- Posible corrupción de datos sensibles o críticos.
- Exposición a ataques de seguridad derivados de estados impredecibles en el flujo del programa.
- Desobediencia al modelo de memoria esperado, lo que puede provocar fallos de hardware o software.

Buenas prácticas

- Declarar los punteros o referencias con el calificador `volatile` cuando apunten a objetos `volatile`.

```
//-- Acceso a objeto volatile a través de puntero no volatile
volatile int hw_reg;
int *ptr = (int *)&hw_reg;
int value = *ptr;
```

```
//-- Acceso a objeto volatile mediante puntero declarado volatile
volatile int hw_reg;
volatile int *ptr = &hw_reg;
int value = *ptr;
```

- No eliminar el calificador `volatile` mediante conversiones de tipo (casting).

```
//-- Eliminación del calificador volatile mediante cast
volatile int status;
int value = *((int *)&status);
```

```
//-- Preservación del calificador volatile sin realizar cast inseguro
volatile int status;
volatile int *pstatus = &status;
int value = *pstatus;
```

- Evitar pasar objetos `volatile` a funciones que esperan parámetros no `volatile`.

```
//-- Paso de objeto volatile a función con parámetro no volatile
void process(int *data);
volatile int sensor;
process((int *)&sensor);
```

```
//-- Definición de la función con parámetro volatile compatible
void process(volatile int *data);
volatile int sensor;
process(&sensor);
```

- Mantener la coherencia del calificador `volatile` en estructuras y sus miembros.

```
//-- Acceso a miembro volatile a través de estructura no volatile
struct Device {
  volatile int control;
};
struct Device dev;
int *p = (int *)&dev.control;
int value = *p;
```

```
//-- Acceso respetando el calificador volatile del miembro
struct Device {
  volatile int control;
};
struct Device dev;
volatile int *p = &dev.control;
int value = *p;
```

2.3.7.2 Estándares metodológicos

- **CERT/C-EXP32**: Do not access a volatile object through a nonvolatile reference.

2.3.8 [c.ini.8] - No modificar objetos con duración temporal

ID	**c.ini.8**	**Uso**	**Recomendado**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los objetos con **duración temporal** son creados automáticamente por el compilador durante la evaluación de expresiones y tienen un tiempo de vida limitado al contexto en el que se generan. Modificar estos objetos produciría un **comportamiento impredecible**, ya que el estándar no garantiza cuándo serán destruidos ni si las modificaciones tendrán efecto.

Codificación insegura

Se muestra una función que devuelve un puntero a un objeto con duración automática (variable local). Después, ese puntero se utiliza para modificar el contenido fuera del ámbito donde el objeto fue creado. El objetivo es entender cómo la vida útil de los objetos afecta a la validez de las referencias y por qué no se debe acceder ni modificar memoria que ya no es válida.

```
#include <stdio.h>
#include <string.h>

//-- Función que devuelve un puntero a una variable local
```

```
char* obtenerMensaje() {
  //-- Variable local con duración automática
  char buffer[32];

  //-- Se copia un texto al buffer
  strcpy(buffer, "Hola Mundo");

  //-- Se retorna la dirección de una variable local (error)
  return buffer;
}
int main() {
  //-- Se obtiene un puntero a memoria que ya no es válida
  char* mensaje = obtenerMensaje();

  //-- Se intenta modificar el contenido apuntado
  mensaje[0] = 'h';

  //-- Se imprime el contenido (comportamiento indefinido)
  printf("%s\n", mensaje);

  return 0;
}
```

Codificación recomendada

Se presenta una alternativa donde la memoria utilizada para almacenar el texto tiene una duración válida más allá del ámbito de la función. En este caso se usa memoria dinámica para asegurar que el objeto sigue existiendo después de retornar, permitiendo su uso y modificación de forma controlada. También se libera la memoria para evitar fugas.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//-- Función que reserva memoria dinámica y retorna un puntero válido
char* obtenerMensaje() {
  //-- Reserva de memoria en el heap
  char* buffer = (char*)malloc(32);

  //-- Verificación de que la reserva fue exitosa
  if (buffer == NULL) {
    return NULL;
  }
  //-- Se copia un texto al buffer
  strcpy(buffer, "Hola Mundo");

  //-- Se retorna un puntero válido
```

```
  return buffer;
}
int main() {
  //-- Se obtiene un puntero a memoria válida
  char* mensaje = obtenerMensaje();

  //-- Verificación antes de usar el puntero
  if (mensaje == NULL) {
    return 1;
  }
  //-- Se modifica el contenido de forma segura
  mensaje[0] = 'h';

  //-- Se imprime el contenido
  printf("%s\n", mensaje);

  //-- Liberación de la memoria reservada
  free(mensaje);

  return 0;
}
```

2.3.8.1 Descripción

La modificación de los objetos con duración temporal presenta un **comportamiento indefinido** según el estándar del lenguaje. Aunque técnicamente es posible obtener la dirección de un objeto temporal y modificar su contenido, esta práctica viola las reglas fundamentales del modelo de memoria y puede producir resultados impredecibles. El problema surge porque el compilador no garantiza la persistencia de estos objetos más allá de su contexto inmediato de uso. Su contenido puede ser sobrescrito, su ubicación en memoria puede cambiar, o pueden ser eliminados en cualquier momento durante la ejecución del programa.

Los riesgos asociados incluyen **corrupción de datos**, comportamiento errático del programa, y fallos difíciles de reproducir que pueden manifestarse de forma intermitente dependiendo de las condiciones de compilación y ejecución.

La mitigación efectiva requiere identificar correctamente cuándo se están manipulando objetos temporales y evitar cualquier intento de modificación directa. Las alternativas incluyen la copia explícita de valores temporales a variables con duración adecuada antes de su manipulación, y el diseño de interfaces que trabajen exclusivamente con objetos persistentes.

Esta regla es especialmente relevante cuando se trabaja con funciones que retornan estructuras por valor, operaciones de conversión de tipos, y expresiones que involucran operadores que generan valores intermedios.

Riesgos asociados

- Condiciones de carrera y comportamientos indefinidos si se modifica un objeto temporal que ya ha sido destruido.
- Vulnerabilidad a corrupción de memoria o heap debido a escrituras en regiones ya liberadas.
- Comportamiento inesperado del programa, incluyendo fallos de segmentación (segmentation faults).
- Dificultad para depurar y mantener el código debido a efectos secundarios no intencionales.
- Exposición a ataques de seguridad (por ejemplo, ejecución de código malicioso o denegación de servicio) aprovechando el uso de datos temporales eliminados.
- Pérdida de la integridad de los datos al modificar objetos cuya vida útil ya terminó.
- Resultados no determinísticos en la ejecución, dificultando la confiabilidad del sistema.

Buenas prácticas

- Evitar modificar estructuras u objetos devueltos por valor desde funciones (temporales).

```
//-- Se modifica directamente el objeto temporal devuelto por la función
get_config().timeout = 30;
```

```
//-- Se asigna el valor a un objeto con almacenamiento propio antes de modificarlo
Config cfg = get_config();
cfg.timeout = 30;
```

- No alterar el contenido de memoria devuelta por funciones que retornan punteros a datos temporales o internos.

```
//-- Se modifica memoria apuntada por un puntero a dato temporal/interno
char *p = get_buffer();
p[0] = 'A';
```

```
//-- Se copia el contenido a un buffer propio antes de modificarlo
char *p = get_buffer();
char local[64];
strncpy(local, p, sizeof(local) - 1);
local[sizeof(local) - 1] = '\0';
local[0] = 'A';
```

- No devolver ni manipular referencias a variables locales con duración automática.

```
//-- Se devuelve la dirección de una variable local con duración automática
char *get_name() {
  char name[16] = "admin";
  return name;
}
```

```
//-- Se utiliza almacenamiento estático o dinámico para evitar duración temporal
char *get_name() {
  static char name[16] = "admin";
  return name;
}
```

2.3.8.2 Estándares metodológicos

- **CERT/C-EXP35**: Do not modify objects with temporary lifetime.

2.4 Expresiones

2.4.1 [c.exp.1] - Garantizar que el valor de una expresión sea independiente del orden de evaluación

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

Las expresiones que contienen múltiples operadores pueden producir **resultados incorrectos** cuando el orden de evaluación no está especificado. Los puntos de secuencia determinan cuándo se completan las operaciones, pero entre ellos el compilador puede reordenar las evaluaciones según criterios de optimización.

Codificación insegura

Se muestra una situación donde se realizan múltiples modificaciones sobre la misma variable dentro de una única expresión. El objetivo es evidenciar cómo el resultado puede variar dependiendo del orden de evaluación que el compilador decida aplicar, generando comportamiento indefinido y resultados impredecibles.

```
#include <stdio.h>

//-- Demostración de modificación múltiple sin orden de evaluación definido
int main(void) {
  int i = 0;
```

```
  int resultado;

  //-- Dos incrementos sobre la misma variable en una sola expresión
  resultado = i++ + i++;

  //-- El valor de 'resultado' e 'i' es indefinido según el estándar
  printf("Valor de i: %d\n", i);
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

Codificación recomendada

Se presenta una forma de separar las operaciones que modifican el estado de una variable en distintas sentencias, garantizando que el resultado no dependa del orden de evaluación. El objetivo es asegurar un comportamiento definido y consistente en cualquier compilador que cumpla el estándar del lenguaje.

```
#include <stdio.h>

//-- Demostración de operaciones separadas con orden de evaluación explícito
int main(void) {
  int i = 0;
  int a;
  int b;
  int resultado;

  //-- Incrementos realizados en sentencias independientes
  a = i;
  i++;

  b = i;
  i++;

  //-- La suma ahora es predecible y definida
  resultado = a + b;

  printf("Valor final de i: %d\n", i);
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

2.4.1.1 Descripción

El orden de evaluación de subexpresiones en una expresión compleja no está definido por el estándar del lenguaje. El compilador puede evaluar los operandos en cualquier secuencia, lo que genera comportamiento indefinido cuando el resultado de la expresión depende de dicho orden.

Esta situación surge cuando una expresión contiene **efectos secundarios** que modifican el mismo objeto o cuando el valor de una variable se lee y modifica dentro de la misma expresión sin puntos de secuencia que garanticen un orden específico.

Los **puntos de secuencia** son ubicaciones en el código donde se garantiza que todas las evaluaciones previas han finalizado y ninguna evaluación posterior ha comenzado. Entre estos puntos, el compilador tiene libertad para reordenar las operaciones según criterios de optimización.

El comportamiento indefinido resultante puede manifestarse de diferentes formas según el compilador, la arquitectura del procesador, el nivel de optimización y otros factores del entorno de compilación. Esto compromete la **portabilidad** del código y puede introducir errores sutiles que son difíciles de detectar durante las pruebas.

La mitigación de este problema requiere estructurar las expresiones de manera que su resultado sea determinista, independientemente del orden de evaluación. Esto implica evitar modificaciones múltiples del mismo objeto dentro de una expresión y separar las operaciones con efectos secundarios en declaraciones independientes cuando sea necesario.

Los compiladores modernos suelen emitir advertencias cuando detectan estas situaciones, aunque no siempre pueden identificar todos los casos problemáticos. El análisis estático de código puede proporcionar detección adicional de patrones que violan esta regla.

Riesgos asociados

- Comportamiento indefinido debido a la evaluación no determinada de expresiones.
- Lectura y/o escritura de una variable antes de que su valor sea actualizado correctamente.
- Vulnerabilidades por acceso concurrente a variables cuando varias expresiones dependen del orden de evaluación.
- Errores difíciles de detectar y reproducir, ya que pueden variar según el compilador o la plataforma.
- Posibilitar condiciones de carrera cuando varias expresiones comparten variables y su modificación depende del orden de evaluación.
- Desbordamiento de buffer o acceso a memoria inválida si las operaciones

dependen de un valor no actualizado.

- Exposición a vulnerabilidades de seguridad como la evasión de controles o la ejecución de código inesperado.
- Dificultad en el mantenimiento y comprensión del código, lo que puede derivar en la introducción de más errores.

Buenas prácticas

- Evitar modificar una variable más de una vez dentro de la misma expresión.

```
//-- Incremento múltiple de la misma variable en una sola expresión
int i = 0;
int resultado = i++ + ++i;
```

```
//-- Separación de operaciones para eliminar dependencia del orden de evaluación
int i = 0;
int temp1 = i;
i++;
i++;
int resultado = temp1 + i;
```

- No combinar efectos secundarios y uso del mismo objeto en argumentos de función.

```
//-- Uso de una variable con efecto secundario en múltiples argumentos
int x = 5;
printf("%d %d", x, x++);
```

```
//-- Evaluación previa de los argumentos antes de la llamada
int x = 5;
int valor = x;
x++;
printf("%d %d", valor, x);
```

- Evitar asignaciones dentro de expresiones donde la variable se reutiliza.

```
//-- Asignación y uso simultáneo en la misma expresión
int a = 3;
int b = (a = 5) + a;
```

```
//-- Separación explícita de la asignación y el cálculo
int a = 3;
a = 5;
int b = a + a;
```

2.4.1.2 Estándares metodológicos

- **MISRA C-12.2**: The value of an expression shall be the same under any order of evaluation that the standard permits.

2.4.2 [c.exp.2] - No depender del orden de evaluación para efectos laterales

ID	c.exp.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las expresiones que contienen **efectos laterales** pueden evaluarse en diferente orden según el compilador y la optimización aplicada. Cuando el resultado de una operación depende del orden específico en que se ejecutan las subexpresiones con efectos laterales, el comportamiento del programa se vuelve **indefinido**. Esta situación genera inconsistencias entre diferentes compiladores y versiones, produciendo errores difíciles de detectar y depurar.

Codificación insegura

Uso de expresiones con efectos laterales múltiples sobre la misma variable dentro de una sola operación. Se busca mostrar cómo ciertas construcciones pueden producir resultados impredecibles debido a que el lenguaje C no define el orden de evaluación de los operandos en muchos casos, lo que puede generar comportamientos indefinidos.

```
#include <stdio.h>

int main() {
  //-- Se declara e inicializa una variable entera
  int i = 5;

  //-- Se usa la variable i múltiples veces con efectos laterales en la misma
expresión
  //-- El orden de evaluación no está definido y el resultado es indefinido
  int result = i++ + ++i;

  //-- Se imprime el resultado calculado y el valor final de i
  printf("Resultado: %d\n", result);
  printf("Valor final de i: %d\n", i);

  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Separación explícita de las operaciones que modifican una variable para evitar depender del orden de evaluación. Se pretende demostrar cómo dividir las expresiones en pasos claros elimina ambigüedades y asegura un comportamiento definido y predecible.

```
#include <stdio.h>

int main() {
  //-- Se declara e inicializa una variable entera
  int i = 5;
  int temp1 = 0;
  int temp2 = 0;
  int result = 0;

  //-- Se separan las operaciones para evitar efectos laterales en la misma
expresión
  temp1 = i;
  i++;

  temp2 = i + 1;
  i++;

  //-- Se calcula el resultado usando valores intermedios ya definidos
  result = temp1 + temp2;

  //-- Se imprime el resultado calculado y el valor final de i
  printf("Resultado: %d\n", result);
  printf("Valor final de i: %d\n", i);

  //-- Fin del programa
  return 0;
}
```

2.4.2.1 Descripción

Esta regla establece que no debemos escribir expresiones que dependan de un orden específico de evaluación cuando existen efectos laterales. Los efectos laterales incluyen modificaciones de variables, operaciones de entrada/salida, y llamadas a funciones que alteran el estado del programa.

El problema surge porque el estándar no define el orden de evaluación para muchos operadores. Los compiladores pueden evaluar los operandos en cualquier secuencia, lo que genera **comportamiento indefinido** cuando el resultado de una expresión cambia según el orden de evaluación elegido.

Los **puntos de secuencia** son momentos específicos durante la ejecución donde se garantiza que todas las evaluaciones previas han terminado y los efectos laterales se han

completado. Entre puntos de secuencia consecutivos, el orden de evaluación permanece sin especificar.

Esta situación presenta riesgos significativos para la seguridad y estabilidad del software. El comportamiento puede variar entre diferentes compiladores, versiones del mismo compilador, o incluso entre compilaciones del mismo código con diferentes opciones de optimización. Esto introduce vulnerabilidades impredecibles y dificulta la detección de errores durante las pruebas.

La mitigación efectiva requiere estructurar las expresiones de manera que el resultado sea independiente del orden de evaluación. Esto implica separar las operaciones con efectos laterales en declaraciones distintas y evitar modificar la misma variable múltiples veces dentro de una expresión compleja. También debemos prestar especial atención a las llamadas de funciones que modifican variables globales o parámetros pasados por referencia.

El cumplimiento de esta regla mejora la **portabilidad del código** y elimina una fuente común de errores sutiles que pueden manifestarse de forma intermitente o dependiente del entorno de compilación.

Riesgos asociados

- Comportamiento indefinido del programa debido a diferentes órdenes de evaluación entre compiladores o arquitecturas.
- Introducción de errores sutiles y difíciles de detectar durante la ejecución en distintas plataformas o con diferentes opciones de compilación.
- Posibles fallos de seguridad por acceso o modificación inesperada de memoria.
- Dependencias ocultas en expresiones que pueden permitir que un atacante manipule el flujo del programa.
- Dificultad para mantener o auditar el código, incrementando el riesgo de introducir vulnerabilidades al modificarlo.
- Resultados inesperados o inconsistentes al combinar operaciones con efectos laterales en la misma expresión.
- Mayor probabilidad de explotar condiciones de carrera o corrupciones de datos globales o estáticos.

Buenas prácticas

- Evitar múltiples efectos laterales en una misma expresión.

```
//-- Incrementos múltiples en una sola expresión generan ambigüedad en el orden de evaluación
int i = 0;
```

```
int a = i++ + i++;
```

```
//-- Separación de efectos laterales en instrucciones independientes
int i = 0;
int temp1 = i;
i++;
int temp2 = i;
i++;
int a = temp1 + temp2;
```

- No modificar una variable más de una vez entre puntos de secuencia.

```
//-- Modificación y lectura sin punto de secuencia definido
int x = 5;
x = x++ + 1;
```

```
//-- Actualización explícita en pasos definidos
int x = 5;
int temp = x;
x++;
x = temp + 1;
```

- No depender del orden de evaluación de argumentos en funciones.

```
//-- Argumentos con efectos laterales cuyo orden no está definido
int i = 1;
printf("%d %d", i++, i++);
```

```
//-- Evaluación previa de argumentos antes de la llamada
int i = 1;
int arg1 = i;
i++;
int arg2 = i;
i++;
printf("%d %d", arg1, arg2);
```

2.4.2.2 Estándares metodológicos

- **CERT/C-EXP30**: Do not depend on the order of evaluation for side effects.

2.5 Estructuras de control y flujo

2.5.1 [c.flow.1] - Asegurar que las construcciones "if/else if" deben terminar con un "else" final

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

Las estructuras de control `if/else if` sin una cláusula `else` final pueden generar comportamientos indefinidos cuando ninguna de las condiciones evaluadas resulta verdadera. Esta omisión introduce **vulnerabilidades de seguridad** y errores lógicos difíciles de detectar, especialmente en sistemas críticos donde todas las posibles rutas de ejecución deben estar explícitamente controladas.

Codificación insegura

Validación de un valor numérico para clasificarlo según su rango. Se muestra una estructura condicional que evalúa distintos casos posibles y cómo el flujo puede quedar incompleto si no se contempla una condición final para cubrir valores no previstos. El objetivo es entender el impacto en la robustez y mantenibilidad del código cuando no se gestionan todos los escenarios.

```
#include <stdio.h>

//-- Programa que clasifica un número sin cubrir todos los casos posibles
int main() {
  int numero;

  //-- Solicita un número al usuario
  printf("Introduce un numero: ");
  scanf("%d", &numero);

  //-- Clasificación incompleta sin else final
  if (numero < 0) {
    printf("El numero es negativo\n");
  } else if (numero == 0) {
    printf("El numero es cero\n");
  } else if (numero > 0 && numero <= 100) {
    printf("El numero es positivo y menor o igual a 100\n");
  }
  //-- Si el numero es mayor a 100 no se ejecuta ningun bloque
  //-- Esto puede generar comportamientos inesperados
  return 0;
}
```

Codificación recomendada

Validación de un valor numérico para clasificarlo según su rango asegurando que todas las posibles condiciones estén contempladas. Se incluye una rama final que actúa como control defensivo, permitiendo manejar valores no previstos y facilitando la detección de errores o cambios futuros en la lógica.

```
#include <stdio.h>

//-- Programa que clasifica un número cubriendo todos los casos posibles
int main() {
  int numero;

  //-- Solicita un número al usuario
  printf("Introduce un numero: ");
  if (scanf("%d", &numero) != 1) {
    //-- Validación básica de entrada
    printf("Entrada no valida\n");
    return 1;
  }
  //-- Clasificación completa con else final
  if (numero < 0) {
    printf("El numero es negativo\n");
  } else if (numero == 0) {
    printf("El numero es cero\n");
  } else if (numero > 0 && numero <= 100) {
    printf("El numero es positivo y menor o igual a 100\n");
  } else {
    //-- Manejo explicito de valores no contemplados previamente
    printf("El numero es mayor que 100\n");
  }
  return 0;
}
```

2.5.1.1 Descripción

Las estructuras condicionales `if/else if` deben incluir siempre una cláusula `else` final que maneje todos los casos no contemplados explícitamente en las condiciones anteriores. Esta práctica garantiza que el programa tenga un comportamiento definido para cualquier valor de entrada, evitando estados indefinidos o comportamientos impredecibles.

La ausencia de una cláusula `else` final puede provocar que el programa no ejecute ninguna acción cuando ninguna de las condiciones se cumple, lo que puede resultar en **variables no inicializadas**, **estados inconsistentes** o **fallos silenciosos** que son difíciles de detectar durante las pruebas. Estos problemas se agravan especialmente cuando las condiciones involucran datos de entrada externos o cálculos complejos donde no todos los casos posibles han sido considerados durante el diseño.

El riesgo aumenta significativamente en sistemas críticos donde cada rama de ejecución debe estar controlada y documentada. Los errores derivados de casos no manejados pueden manifestarse como **vulnerabilidades de seguridad**, **corrupción de datos** o **comportamientos erráticos** que comprometen la fiabilidad del sistema.

La mitigación de estos riesgos se logra implementando una cláusula `else` que contenga una acción apropiada para el contexto específico. Esta acción puede incluir la asignación de valores por defecto, el registro de eventos inesperados, la activación de mecanismos de recuperación de errores, o la terminación controlada del programa. La elección de la estrategia depende de los requisitos funcionales y de seguridad del sistema.

Esta regla se alinea con los principios de **programación defensiva** y **manejo explícito de errores**, contribuyendo a la creación de código más robusto y predecible. Su aplicación sistemática facilita además las tareas de mantenimiento y depuración al hacer explícitos todos los caminos de ejecución posibles.

Riesgos asociados

- Comportamiento inesperado al no contemplar todos los caminos posibles en la lógica de decisión.
- Condiciones no manejadas que pueden derivar en fallas silenciosas o resultados incorrectos.
- Exposición a vulnerabilidades de seguridad, ya que los casos no tratados pueden ser explotados.
- Dificultad para mantener y auditar el código, lo que complica la identificación de errores.
- Incremento en la posibilidad de errores lógicos (logical bugs) y paths no alcanzables.
- Fuga de información sensible si la omisión del “else” permite retornos inesperados o saltos de flujo indeseados.
- Desincronización en la gestión de recursos (por ejemplo, memoria o archivos abiertos) si algunos caminos no garantizan liberación/cierre.
- Aumento de la deuda técnica al producir código menos claro y más propenso a equivocaciones futuras.

Buenas prácticas

- Incluir siempre un bloque `else` final para manejar casos no contemplados explícitamente.

```
//-- Estructura if/else if sin else final
int validar(int tipo) {
  if (tipo == 1) {
```

```
    procesar_tipo1();
  } else if (tipo == 2) {
    procesar_tipo2();
  }
}
```

```
//-- Estructura if/else if con else final para caso no previsto
int validar(int tipo) {
  if (tipo == 1) {
    procesar_tipo1();
  } else if (tipo == 2) {
    procesar_tipo2();
  } else {
    registrar_error("Tipo no valido");
  }
}
```

- Usar el `else` final para aplicar una acción defensiva como registrar, abortar o retornar error.

```
//-- Rama no contemplada sin tratamiento explicito
int obtener_permiso(int rol) {
  if (rol == ADMIN) {
    return 1;
  } else if (rol == USER) {
    return 0;
  }
}
```

```
//-- Rama no contemplada devuelve error por defecto
int obtener_permiso(int rol) {
  if (rol == ADMIN) {
    return 1;
  } else if (rol == USER) {
    return 0;
  } else {
    return -1;
  }
}
```

- Utilizar el `else` final para garantizar un valor de retorno definido en todas las rutas de ejecución.

```
//-- Funcion sin retorno garantizado en todos los caminos
int calcular(int opcion) {
  if (opcion == 10) {
    return 100;
```

```
  } else if (opcion == 20) {
    return 200;
  }
}
```

```
//-- Funcion con retorno garantizado mediante else final
int calcular(int opcion) {
  if (opcion == 10) {
    return 100;
  } else if (opcion == 20) {
    return 200;
  } else {
    return 0;
  }
}
```

2.5.2 [c.flow.2] - Asegurar que las sentencias switch deben tener una estructura completa y controlada

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

Las sentencias `switch` incompletas o mal estructuradas pueden provocar comportamientos impredecibles en el programa. La ausencia de casos `default`, la falta de sentencias `break` o el manejo inadecuado de los valores de entrada pueden generar **ejecución no intencionada** de código y vulnerabilidades de seguridad.

Codificación insegura

Gestión de un menú simple usando una sentencia `switch` para ejecutar distintas acciones según la opción ingresada por el usuario. Se pretende mostrar la importancia de controlar todos los casos posibles, evitar ejecuciones no deseadas entre ramas y asegurar una estructura clara y predecible del flujo.

```
#include <stdio.h>

int main() {
  //-- Declaración de la opción seleccionada por el usuario
  int opcion;

  //-- Solicita una opción al usuario
  printf("Seleccione una opcion (1-3): ");
  scanf("%d", &opcion);
```

```
  //-- Estructura switch sin control completo de casos
  switch (opcion) {
    case 1:
      //-- Acción para la opción 1
      printf("Ha seleccionado la opcion 1\n");
    case 2:
      //-- Acción para la opción 2
      printf("Ha seleccionado la opcion 2\n");
      break;
    case 3:
      //-- Acción para la opción 3
      printf("Ha seleccionado la opcion 3\n");
      break;
  }
  //-- Finalización del programa
  return 0;
}
```

Codificación recomendada

Gestión de un menú simple usando una sentencia `switch` con una estructura completa, incluyendo control explícito de cada caso, uso adecuado de `break` para evitar ejecuciones no intencionadas y una rama `default` para manejar valores no esperados. Se busca reforzar un flujo de ejecución claro, seguro y mantenible.

```
#include <stdio.h>

int main() {
  //-- Declaración de la opción seleccionada por el usuario
  int opcion;

  //-- Solicita una opción al usuario
  printf("Seleccione una opcion (1-3): ");
  if (scanf("%d", &opcion) != 1) {
    //-- Manejo básico de error de entrada
    printf("Entrada no valida\n");
    return 1;
  }
  //-- Estructura switch con control completo y explícito
  switch (opcion) {
    case 1:
      //-- Acción para la opción 1
      printf("Ha seleccionado la opcion 1\n");
      break;
    case 2:
      //-- Acción para la opción 2
      printf("Ha seleccionado la opcion 2\n");
      break;
```

```
    case 3:
      //-- Acción para la opción 3
      printf("Ha seleccionado la opcion 3\n");
      break;
    default:
      //-- Manejo explícito de valores no contemplados
      printf("Opcion no valida\n");
      break;
  }
  //-- Finalización del programa
  return 0;
}
```

2.5.2.1 Descripción

Las sentencias `switch` requieren una estructura completa que incluya el manejo explícito de todos los casos posibles y la presencia obligatoria de una cláusula `default`. Esta regla establece que cada rama `case` debe terminar con una sentencia de control de flujo apropiada, típicamente `break`, para evitar la ejecución no intencionada de código en casos subsiguientes.

La ausencia de terminadores explícitos en las ramas `case` provoca el fenómeno conocido como **fall-through**, donde la ejecución continúa hacia el siguiente caso sin evaluación de condiciones. Este comportamiento puede generar resultados inesperados y errores lógicos difíciles de detectar durante las pruebas.

La omisión de la cláusula `default` representa un riesgo adicional, ya que valores no contemplados en los casos explícitos pueden provocar comportamientos indefinidos o la ejecución de código no relacionado. Esta situación es particularmente problemática cuando se trabaja con tipos de datos que pueden asumir valores fuera del rango esperado.

La mitigación de estos riesgos se logra mediante la implementación sistemática de terminadores de flujo en cada rama `case` y la inclusión obligatoria de una cláusula `default` que maneje explícitamente los casos no previstos. Cuando el fall-through sea intencionado, debe documentarse claramente mediante comentarios que indiquen la intencionalidad del comportamiento.

Las herramientas de análisis estático pueden detectar automáticamente sentencias `switch` incompletas y casos sin terminadores apropiados, facilitando la identificación temprana de estas vulnerabilidades durante el proceso de desarrollo.

Riesgos asociados

- Ejecución de casos imprevistos por ausencia de sentencia `break` (fall through), causando comportamientos inesperados.
- Pérdida del control de flujo si no se incluye un caso `default`, dejando situaciones

sin manejar.

- Vulnerabilidad ante ataques de escalada de privilegios o ejecución arbitraria de código al procesar valores no contemplados.
- Errores lógicos difíciles de detectar y depurar, aumentando la probabilidad de fallos en producción.
- Datos corruptos y estados inconsistentes por ejecución de bloques de código no deseados.
- Aumento en la superficie de ataque al no validar adecuadamente todos los casos posibles.
- Fuga de información sensible si el flujo accidentalmente toca código que maneja datos sensibles.

Buenas prácticas

- Incluir siempre una cláusula `default` para manejar valores no previstos explícitamente.

```
//-- Manejo incompleto sin caso por defecto
switch(opcion) {
  case 1: procesar(); break;
  case 2: cancelar(); break;
}
```

```
//-- Manejo completo incluyendo validación por defecto
switch(opcion) {
  case 1: procesar(); break;
  case 2: cancelar(); break;
  default: registrar_error(); break;
}
```

- Finalizar cada `case` con `break` para evitar ejecuciones no intencionadas por caída (fall-through).

```
//-- Omisión de break que provoca ejecución no deseada
switch(estado) {
  case 0: iniciar();
  case 1: detener(); break;
}
```

```
//-- Separación explícita de cada caso con break
switch(estado) {
  case 0: iniciar(); break;
  case 1: detener(); break;
```

```
}
```

- Validar previamente el rango del valor evaluado antes de entrar al `switch` cuando provenga de fuentes externas.

```
//-- Uso directo de valor externo sin validación
int comando = atoi(entrada);
switch(comando) {
  case 1: ejecutar(); break;
  case 2: salir(); break;
  default: error(); break;
}
```

```
//-- Validación previa del rango permitido
int comando = atoi(entrada);
if(comando < 1 || comando > 2) {
  error();
} else {
  switch(comando) {
    case 1: ejecutar(); break;
    case 2: salir(); break;
  }
}
```

- Agrupar múltiples casos relacionados de forma explícita y controlada para mejorar claridad y evitar ambigüedades.

```
//-- Casos múltiples sin estructura clara
switch(tipo) {
  case 1:
  case 2: procesar_tipo_basico(); break;
  case 3: procesar_tipo_avanzado(); break;
}
```

```
//-- Agrupación documentada y controlada de casos equivalentes
switch(tipo) {
  case 1:
  case 2: {
    procesar_tipo_basico();
    break;
  }
  case 3: procesar_tipo_avanzado(); break;
  default: registrar_error(); break;
}
```

2.5.3 [c.flow.3] - Evitar utilizar "goto" o asegurar que solo salta a etiquetas dentro del bloque envolvente

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

El uso inadecuado de `goto` puede generar **flujos de control impredecibles** que comprometen la legibilidad y mantenibilidad del código. Los saltos a etiquetas ubicadas fuera del bloque actual pueden eludir inicializaciones de variables y crear estructuras de control difíciles de analizar y depurar.

Codificación insegura

Control de flujo usando `goto` para saltar a una etiqueta ubicada dentro de un bloque interno. Se muestra cómo este patrón dificulta la lectura, puede romper la estructura lógica del programa y aumenta el riesgo de errores al alterar el flujo natural de ejecución.

```
#include <stdio.h>

//-- Demostración de uso de goto hacia una etiqueta dentro de un bloque interno
int main(void) {
  int value = 0;

  //-- Salto directo a una etiqueta dentro de un bloque if
  goto inner_label;

  if (value == 0) {
    //-- Código que podría quedar omitido o ejecutarse fuera de orden
    printf("Dentro del bloque if antes de la etiqueta\n");

    inner_label:
    //-- Punto de entrada artificial dentro del bloque
    printf("Etiqueta alcanzada con goto\n");
  }
  //-- Flujo continúa después del bloque
  printf("Fin del programa\n");
  return 0;
}
```

Codificación recomendada

Control de flujo estructurado utilizando sentencias condicionales en lugar de `goto`. Se mantiene la ejecución dentro de los bloques correspondientes, mejorando la legibilidad, mantenibilidad y reduciendo riesgos asociados a saltos arbitrarios.

```c
#include <stdio.h>

//-- Demostración de control de flujo estructurado sin goto
int main(void) {
  int value = 0;

  //-- Evaluación normal de la condición
  if (value == 0) {
    //-- Código ejecutado de forma clara y predecible
    printf("Dentro del bloque if\n");
  }
  //-- Flujo continúa de manera estructurada
  printf("Fin del programa\n");
  return 0;
}
```

2.5.3.1 Descripción

La instrucción `goto` permite realizar saltos incondicionales que compromete la legibilidad del código y aumenta significativamente la complejidad del análisis estático. Esta práctica dificulta el mantenimiento, la depuración y la verificación formal del software, especialmente en sistemas críticos donde la trazabilidad del flujo de control resulta esencial.

Los saltos hacia atrás mediante `goto` pueden generar bucles implícitos que no siguen las estructuras de control convencionales, complicando el análisis de terminación y la detección de bucles infinitos. Los saltos hacia adelante que omiten inicializaciones de variables o liberación de recursos pueden provocar **estados inconsistentes** y fugas de memoria.

Cuando el uso de `goto` resulte inevitable, la restricción de los saltos únicamente a etiquetas ubicadas dentro del mismo bloque de código envolvente mantiene un nivel mínimo de estructura y predictibilidad. Esta limitación preserva el contexto local y reduce la posibilidad de generar flujos de control problemáticos.

La mitigación más efectiva consiste en emplear **estructuras de control estructuradas** como bucles `for`, `while` y `do-while`, junto con condicionales `if-else` y `switch`. Para el manejo de errores y limpieza de recursos, se pueden utilizar patrones de salida única con variables de estado o funciones auxiliares que encapsulen la lógica de limpieza.

Riesgos asociados

- Introducción de vulnerabilidades por saltos incontrolados en el flujo de ejecución.
- Dificultad para entender y mantener el código, aumentando la probabilidad de errores que pueden ser explotados.

- Salto inadvertido sobre inicialización o liberación de recursos, provocando fugas de memoria o uso de variables sin inicializar.
- Alteración inesperada del flujo de control, facilitando la ejecución de código inseguro o malicioso.
- Omisión de controles de seguridad o validaciones necesarias al saltar entre bloques no anidados.
- Posibilidad de corromper el estado interno del programa, generando condiciones de carrera o inconsistencias.
- Dificultad para aplicar análisis estático de seguridad y herramientas automáticas, ya que el flujo se vuelve impredecible.

Buenas prácticas

- Evitar el uso de `goto` sustituyéndolo por estructuras de control como bucles o condicionales.

```
//-- Uso de goto para controlar el flujo del bucle
int i = 0;
inicio:
if (i < 5) {
  printf("%d\n", i);
  i++;
  goto inicio;
}
```

```
//-- Uso de estructura while para controlar el flujo
int i = 0;
while (i < 5) {
  printf("%d\n", i);
  i++;
}
```

- Si se utiliza `goto`, asegurarse de que salta únicamente a etiquetas dentro del mismo bloque envolvente.

```
//-- Salto a etiqueta fuera del bloque, dificultando el seguimiento del flujo
if (cond) {
  goto fuera;
}
printf("Dentro del bloque\n");
fuera:
printf("Etiqueta fuera\n");
```

```
//-- Salto a etiqueta dentro del mismo bloque envolvente
```

```
if (cond) {
  goto fin;
  printf("Codigo intermedio\n");
  fin:
  printf("Salida controlada\n");
}
```

- Utilizar `goto` únicamente para manejo centralizado de errores y liberación de recursos en funciones complejas. Aun así, intentar evitarlo.

```
//-- Multiples puntos de retorno sin liberacion centralizada
FILE *f = fopen("data.txt", "r");
if (!f) return;
char *buf = malloc(100);
if (!buf) return;
if (error) return;
free(buf);
fclose(f);
```

```
//-- Uso de goto para limpieza centralizada de recursos
FILE *f = fopen("data.txt", "r");
if (!f) return;
char *buf = malloc(100);
if (!buf) goto cleanup_file;
if (error) goto cleanup;
cleanup:
  free(buf);
cleanup_file:
  fclose(f);
```

2.5.4 [c.flow.4] - Evitar el uso de llamadas recursivas directas o indirectas

ID	c.flow.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Design

Las llamadas recursivas pueden provocar **desbordamiento de pila** cuando la profundidad de recursión excede la capacidad disponible. Este problema se agrava en sistemas embebidos con memoria limitada y cuando no existen mecanismos adecuados de control de profundidad, pudiendo resultar en fallos del sistema o comportamientos impredecibles.

Codificación insegura

Se muestra una implementación que calcula el factorial de un número utilizando llamadas recursivas directas. La intención es ilustrar cómo la recursividad puede provocar consumo excesivo de pila, riesgo de desbordamiento y dificultad para controlar los límites cuando no se validan adecuadamente las entradas.

```
#include <stdio.h>
#include <stdlib.h>

int factorial(int n) {
  //-- caso base para detener la recursion
  if (n <= 1) {
    return 1;
  }
  //-- llamada recursiva directa sin control de profundidad
  return n * factorial(n - 1);
}
int main(int argc, char *argv[]) {
  //-- verificacion basica de argumentos
  if (argc != 2) {
    printf("Uso: %s <numero>\n", argv[0]);
    return EXIT_FAILURE;
  }
  //-- conversion directa sin validar rango
  int numero = atoi(argv[1]);

  //-- posible desbordamiento de pila si el numero es grande
  int resultado = factorial(numero);

  printf("Factorial de %d es %d\n", numero, resultado);
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Se presenta una versión que calcula el factorial utilizando una solución iterativa. El objetivo es evitar llamadas recursivas directas o indirectas, reduciendo el riesgo de desbordamiento de pila y permitiendo un mejor control de validaciones y límites en tiempo de ejecución.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main(int argc, char *argv[]) {
  //-- verificacion de numero correcto de argumentos
  if (argc != 2) {
```

```
    printf("Uso: %s <numero>\n", argv[0]);
    return EXIT_FAILURE;
  }
  //-- conversion y validacion basica de rango
  long numero = strtol(argv[1], NULL, 10);
  if (numero < 0 || numero > 20) {
    //-- limite para evitar desbordamiento de entero
    printf("Numero fuera de rango permitido (0-20)\n");
    return EXIT_FAILURE;
  }
  //-- calculo iterativo sin recursion
  unsigned long long resultado = 1;
  for (long i = 1; i <= numero; i++) {
    resultado *= i;
  }
  printf("Factorial de %ld es %llu\n", numero, resultado);
  return EXIT_SUCCESS;
}
```

2.5.4.1 Descripción

Las llamadas recursivas directas o indirectas pueden provocar el agotamiento del stack de ejecución cuando no se controlan adecuadamente. Una función recursiva directa se invoca a sí misma, mientras que la recursión indirecta ocurre cuando una función llama a otra que eventualmente invoca la función original, creando un ciclo de llamadas.

El principal riesgo radica en que cada llamada recursiva consume espacio en el stack para almacenar variables locales, parámetros y direcciones de retorno. Sin mecanismos de control apropiados, la recursión puede continuar hasta agotar completamente la memoria disponible del stack, resultando en **stack overflow** y terminación abrupta del programa.

La recursión indirecta presenta desafíos adicionales de detección y análisis, ya que el ciclo de llamadas puede involucrar múltiples funciones y no resulta evidente durante la revisión del código. Este tipo de recursión puede introducirse inadvertidamente durante el mantenimiento o la refactorización del código.

Los sistemas embebidos y aplicaciones críticas son particularmente vulnerables debido a las limitaciones estrictas de memoria y la necesidad de comportamiento predecible. En estos entornos, el agotamiento del stack puede comprometer la estabilidad del sistema completo.

La mitigación efectiva incluye el reemplazo de algoritmos recursivos por implementaciones iterativas equivalentes, el establecimiento de límites máximos de profundidad de recursión, y la validación exhaustiva de condiciones de terminación. El análisis estático del código puede identificar patrones de recursión potencialmente problemáticos durante el desarrollo.

Las herramientas de análisis de flujo de control ayudan a detectar ciclos de llamadas indirectas que podrían no ser evidentes durante la inspección manual del código. La documentación clara de las dependencias entre funciones facilita la identificación de posibles rutas recursivas.

Riesgos asociados

- Desbordamiento de pila (stack overflow) debido a la profundidad no controlada de llamadas recursivas.
- Condiciones de denegación de servicio (DoS) si la recursividad entra en un ciclo infinito o excesivo.
- Dificultad en el análisis y control del flujo de ejecución, favoreciendo la introducción de errores lógicos y vulnerabilidades.
- Consumo excesivo de recursos del sistema, degradando el rendimiento de la aplicación y potencialmente afectando otros procesos.
- Dificultad para predecir el comportamiento ante entradas inesperadas o maliciosas, incrementando la superficie de ataque.
- Generación de trazas de error poco claras que dificultan la detección y remediación de fallos de seguridad.
- Posible corrupción de memoria si los parámetros, variables locales o el stack-frame superan los límites permitidos.

Buenas prácticas

- Reemplazar llamadas recursivas directas por estructuras iterativas controladas (`while`, `for`) para evitar desbordamiento de pila.

```
//-- Uso de bucle iterativo para evitar crecimiento de pila
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

```
//-- Implementación iterativa con control explícito
int factorial(int n) {
  int resultado = 1;
  for (int i = 2; i <= n; i++) {
    resultado *= i;
  }
  return resultado;
}
```

- Evitar recursividad indirecta entre funciones que pueda dificultar el análisis del

flujo y provocar ciclos no controlados.

```
//-- Llamadas cruzadas que pueden generar ciclo infinito
void funcionA(int n);
void funcionB(int n);
void funcionA(int n) {
  if (n > 0) funcionB(n - 1);
}
void funcionB(int n) {
  if (n > 0) funcionA(n - 1);
}
```

```
//-- Sustitución por bucle con control centralizado
void procesar(int n) {
  while (n > 0) {
    n--;
  }
}
```

- Si no es posible eliminar la recursividad, implementar límites estrictos de profundidad para mitigar riesgos de desbordamiento de pila.

```
//-- Recursión sin límite explícito de profundidad
void procesar(int nivel) {
  procesar(nivel + 1);
}
```

```
//-- Control explícito de profundidad máxima
#define MAX_NIVEL 100
void procesar(int nivel) {
  if (nivel >= MAX_NIVEL) return;
  procesar(nivel + 1);
}
```

- Preferir estructuras de datos explícitas (pilas manuales) para simular recursión cuando se requiera control detallado de memoria.

```
//-- Recorrido recursivo de estructura enlazada
struct Nodo {
  int valor;
  struct Nodo* siguiente;
};
void recorrer(struct Nodo* nodo) {
  if (nodo == NULL) return;
  recorrer(nodo->siguiente);
}
```

```
//-- Uso de pila explícita para controlar memoria
void recorrer(struct Nodo* nodo) {
  struct Nodo* pila[100];
  int tope = 0;
  while (nodo != NULL && tope < 100) {
    pila[tope++] = nodo;
    nodo = nodo->siguiente;
  }
}
```

2.6 Funciones

2.6.1 [c.func.1] - Llamar a funciones con el número y tipo correctos de argumentos

ID	c.func.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones esperan recibir argumentos con tipos y cantidad específicos según su declaración. Cuando se invoca una función con argumentos incorrectos en tipo o número, se produce un **comportamiento impredecible** que puede generar corrupción de datos, fallos de segmentación o vulnerabilidades explotables. Este problema es especialmente crítico en funciones variádicas como `printf` y en llamadas donde los tipos no coinciden exactamente con los parámetros esperados.

Codificación insegura

Uso de una función para multiplicar dos números donde no se define correctamente el prototipo y se realiza la llamada con un número y tipo de argumentos distinto al esperado. Se busca observar cómo una declaración incompleta permite compilar el programa, pero genera un comportamiento indefinido en tiempo de ejecución.

```
#include <stdio.h>

//-- Declaración incompleta de la función, sin especificar parámetros
int multiply();

int main(void) {
  //-- Llamada con un solo argumento y de tipo double
  double value = 5.5;
  int result = multiply(value);

  //-- Impresión del resultado (comportamiento indefinido)
  printf("Resultado: %d\n", result);
```

```
  return 0;
}
//-- Definición real con dos parámetros de tipo int
int multiply(int a, int b) {
  //-- Multiplica dos enteros
  return a * b;
}
```

Codificación recomendada

Uso de una función para multiplicar dos números donde se declara el prototipo con el número y tipo exacto de parámetros, y la llamada coincide con dicha definición. Se busca mostrar cómo la validación en compilación previene errores y comportamientos indefinidos.

```
#include <stdio.h>

//-- Declaración correcta con número y tipo de parámetros definidos
int multiply(int a, int b);

int main(void) {
  //-- Variables enteras que coinciden con la definición
  int value1 = 5;
  int value2 = 4;

  //-- Llamada con el número y tipo correcto de argumentos
  int result = multiply(value1, value2);

  //-- Impresión del resultado esperado
  printf("Resultado: %d\n", result);
  return 0;
}
//-- Definición que coincide exactamente con el prototipo
int multiply(int a, int b) {
  //-- Multiplica dos enteros de forma segura
  return a * b;
}
```

2.6.1.1 Descripción

Esta regla establece que las llamadas a funciones deben realizarse con el número exacto de parámetros que la función espera y con los tipos de datos correctos para cada parámetro. El compilador debe poder verificar la correspondencia entre los argumentos proporcionados y los parámetros declarados en la función.

La verificación de tipos y número de argumentos depende de que el compilador tenga acceso a la **declaración completa de la función** antes del punto de llamada. Sin esta

información, el compilador no puede detectar discrepancias entre lo que la función espera y lo que recibe, pudiendo resultar un comportamiento indefinido durante la ejecución.

Los problemas más comunes incluyen pasar un número incorrecto de argumentos, utilizar tipos incompatibles sin conversión explícita, o realizar llamadas a funciones sin haber incluido los headers correspondientes. Estas situaciones pueden provocar **corrupción de la pila**, acceso a memoria no válida, o interpretación incorrecta de los datos pasados como parámetros.

El riesgo se incrementa significativamente cuando se trabaja con funciones que tienen parámetros de diferentes tamaños o cuando se utilizan punteros. La falta de correspondencia puede causar que la función lea o escriba en ubicaciones de memoria incorrectas, comprometiendo tanto la integridad de los datos como la estabilidad del programa.

La mitigación principal consiste en asegurar que todas las funciones estén **correctamente declaradas** antes de su uso, típicamente mediante la inclusión de los archivos de cabecera apropiados. El uso de prototipos de función completos permite al compilador realizar verificaciones automáticas y generar advertencias o errores cuando detecta inconsistencias.

Las herramientas de análisis estático pueden identificar muchos de estos problemas durante la compilación, especialmente cuando se configuran con niveles altos de advertencias. La activación de flags como `-Wall` y `-Wextra` en compiladores como GCC ayuda a detectar llamadas problemáticas antes de que el código llegue a producción.

Riesgos asociados

- Corromper la pila (stack) debido a desajuste en el número de parámetros, provocando fallos de segmentación (segmentation fault).
- Lectura/escritura fuera de límites en memoria (buffer overflow), abriendo la puerta a vulnerabilidades de ejecución de código arbitrario.
- Comportamiento indefinido, dificultando el mantenimiento y depuración del código.
- Divulgación accidental de información sensible al retornar o imprimir datos de áreas de memoria incorrectas.
- Riesgo incrementado de ataques de explotación (por ejemplo, return-oriented programming) debido a desbordamientos de pila o de heap causados por argumentos incorrectos.
- Incompatibilidad con futuras actualizaciones de bibliotecas o sistemas si las llamadas no cumplen los prototipos estándar.
- Bloqueo (crash) de la aplicación, lo que puede provocar denegación de servicio

(DoS).

- Ejecución de operaciones lógicas erróneas, generando errores funcionales silenciosos y potencialmente fraudulentos.

Buenas prácticas

- Verificar siempre la firma declarada de la función antes de invocarla para asegurar coincidencia exacta de tipos y cantidad de argumentos.

```
//-- Llamada con número incorrecto de argumentos
int suma(int a, int b);
int r = suma(5);
```

```
//-- Llamada con número correcto de argumentos según la declaración
int suma(int a, int b);
int r = suma(5, 10);
```

- Evitar conversiones implícitas peligrosas asegurando que los tipos de los argumentos coincidan exactamente con los parámetros esperados.

```
//-- Paso de tipo incompatible que provoca conversión implícita
void procesar(int valor);
double dato = 3.14;
procesar(dato);
```

```
//-- Conversión explícita o ajuste del tipo antes de la llamada
void procesar(int valor);
double dato = 3.14;
procesar((int)dato);
```

- Declarar prototipos de funciones antes de su uso para permitir la verificación de tipos en tiempo de compilación.

```
//-- Llamada sin prototipo previo visible
int resultado = multiplicar(2, 3);
int multiplicar(int a, int b) {
  return a * b;
}
```

```
//-- Prototipo declarado antes de la llamada
int multiplicar(int a, int b);
int resultado = multiplicar(2, 3);
int multiplicar(int a, int b) {
  return a * b;
}
```

- Respetar el tipo exacto de punteros esperado por la función para evitar comportamiento indefinido.

```
//-- Paso de puntero de tipo incompatible
void imprimirCadena(char *texto);
int numero = 100;
imprimirCadena((char *)&numero);
```

```
//-- Uso de puntero con tipo correcto según la firma
void imprimirCadena(char *texto);
char mensaje[] = "Hola";
imprimirCadena(mensaje);
```

- En funciones variádicas, proporcionar argumentos coherentes con el formato especificado para evitar accesos indebidos a memoria.

```
//-- Argumento no coincide con el especificador de formato
printf("%d", "texto");
```

```
//-- Argumento coincide con el especificador de formato
printf("%s", "texto");
```

2.6.1.2 Estándares metodológicos

- **CERT/C-EXP37**: Call functions with the correct number and type of arguments.

2.6.2 [c.func.2] - No llamar "va_arg" con un tipo incorrecto

ID	c.func.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La función `va_arg` requiere que el tipo especificado coincida exactamente con el tipo del argumento variable correspondiente. Un desajuste entre estos tipos puede provocar corrupción de datos o fallos en tiempo de ejecución, especialmente en arquitecturas donde diferentes tipos tienen distintos tamaños o alineaciones en memoria.

Codificación insegura

Uso de funciones con argumentos variables para mostrar cómo una lectura con tipo incorrecto genera comportamiento indefinido, provocando resultados inesperados o fallos en tiempo de ejecución. Se pretende observar qué ocurre cuando el tipo real

pasado a la función no coincide con el tipo utilizado en la macro de acceso a los argumentos.

```
#include <stdio.h>
#include <stdarg.h>

//-- Función que recibe un número variable de argumentos y los interpreta como
enteros
void imprimir_entero(const char *formato, ...) {
  va_list args;

  //-- Inicializa la lista de argumentos variables
  va_start(args, formato);

  //-- Se obtiene el siguiente argumento como si fuera int
  int valor = va_arg(args, int);

  printf("Valor interpretado como int: %d\n", valor);

  //-- Finaliza el uso de la lista
  va_end(args);
}
int main(void) {
  //-- Se pasa un double, pero la función lo leerá como int
  double numero = 3.14159;

  imprimir_entero("d", numero);

  return 0;
}
```

Codificación recomendada

Uso correcto de funciones con argumentos variables asegurando que el tipo real del argumento coincide exactamente con el tipo especificado en la llamada a la macro de acceso. Se pretende demostrar cómo mantener la coherencia de tipos para evitar comportamiento indefinido.

```
#include <stdio.h>
#include <stdarg.h>

//-- Función que recibe un número variable de argumentos y los interpreta como
double
void imprimir_double(const char *formato, ...) {
  va_list args;

  //-- Inicializa la lista de argumentos variables
  va_start(args, formato);
```

```
  //-- Se obtiene el siguiente argumento con el tipo correcto
  double valor = va_arg(args, double);

  printf("Valor interpretado como double: %f\n", valor);

  //-- Finaliza el uso de la lista
  va_end(args);
}
int main(void) {
  //-- Se pasa un double y se lee como double
  double numero = 3.14159;

  imprimir_double("f", numero);

  return 0;
}
```

2.6.2.1 Descripción

La función `va_arg` extrae argumentos de una lista de argumentos variables y requiere que el tipo especificado coincida exactamente con el tipo del argumento real. Cuando se proporciona un tipo incorrecto, el comportamiento del programa se vuelve indefinido, ya que la función interpreta los bytes de memoria según el tipo especificado sin realizar conversiones automáticas.

Esta discrepancia entre tipos puede producirse de varias formas: especificar un tipo de diferente tamaño al argumento real, usar tipos con diferentes representaciones en memoria, o aplicar promociones de tipos incorrectas. Los argumentos pasados a funciones variádicas sufren promociones automáticas según las reglas del lenguaje, y `va_arg` debe usar el tipo promovido correspondiente.

Los riesgos asociados incluyen **corrupción de datos** al interpretar incorrectamente los valores, **accesos a memoria no válidos** cuando los tamaños de tipos difieren, y **fallos de segmentación** en casos severos. Estos problemas son especialmente peligrosos porque pueden manifestarse de forma intermitente o en configuraciones específicas.

La mitigación efectiva requiere un conocimiento preciso de las reglas de promoción de tipos y la implementación de mecanismos de validación. Las técnicas incluyen el uso de especificadores de formato para determinar tipos esperados, la implementación de sistemas de tipos explícitos mediante enumeraciones o constantes, y la validación de correspondencia entre argumentos y tipos. Las herramientas de análisis estático pueden detectar algunas instancias de este problema, aunque la naturaleza dinámica de las funciones variádicas limita su efectividad.

Riesgos asociados

- Lectura de datos corruptos o inesperados de la pila de argumentos.

- Desbordamiento del buffer al interpretar el argumento como un tipo de mayor tamaño.
- Acceso a memoria inválida, lo que puede derivar en fallos de segmentación (segmentation fault).
- Pérdida de integridad de los datos debido a interpretación incorrecta de los valores.
- Comportamiento indefinido de la aplicación, dificultando la detección y solución de errores.
- Potenciales vulnerabilidades de seguridad, como ejecución de código arbitrario o divulgación de información sensible.
- Incompatibilidad entre múltiples plataformas o compiladores debido a diferencias en la representación de tipos en memoria.
- Dificultad en el mantenimiento y revisión del código originados por errores sutiles y ocultos.

Buenas prácticas

- Utilizar en `va_arg` exactamente el mismo tipo que fue promovido y pasado a la función variádica.

```
//-- Uso de tipo distinto al realmente pasado (int vs double)
void f(int n, ...) {
  va_list ap;
  va_start(ap, n);
  int valor = va_arg(ap, int);
  va_end(ap);
}
```

```
//-- Uso del mismo tipo con el que se pasó el argumento (double)
void f(int n, ...) {
  va_list ap;
  va_start(ap, n);
  double valor = va_arg(ap, double);
  va_end(ap);
}
```

- Considerar las promociones por defecto (por ejemplo, `float` se promueve a `double`, `char` y `short` a `int`).

```
//-- Lectura como float cuando el argumento fue promovido a double
void f(int n, ...) {
  va_list ap;
  va_start(ap, n);
```

```
  float x = va_arg(ap, float);
  va_end(ap);
}
```

```
//-- Lectura como double respetando la promoción por defecto
void f(int n, ...) {
  va_list ap;
  va_start(ap, n);
  double x = va_arg(ap, double);
  va_end(ap);
}
```

- Definir y documentar claramente el contrato de tipos esperados en funciones variádicas para evitar interpretaciones inconsistentes.

```
//-- Interpretación implícita del tipo sin contrato claro
void log_value(const char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  int v = va_arg(ap, int);
  va_end(ap);
}
```

```
//-- Uso de indicador explícito para determinar el tipo esperado
void log_value(int tipo, ...) {
  va_list ap;
  va_start(ap, tipo);
  if (tipo == 1) {
    int v = va_arg(ap, int);
  }
  va_end(ap);
}
```

2.6.2.2 Estándares metodológicos

- **CERT/C-EXP47**: Do not call va_arg with an argument of the incorrect type.

2.6.3 [c.func.3] - No llamar a "system"

ID	c.func.3	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La función `system()` presenta **vulnerabilidades de seguridad críticas** al ejecutar comandos del sistema operativo. Su uso permite ataques de inyección de comandos

cuando procesa entradas no validadas, comprometiendo la integridad del sistema. Además, introduce dependencias del entorno y comportamientos impredecibles entre plataformas.

Codificación insegura

Uso de la función `system()` para ejecutar comandos del sistema operativo construyendo la orden a partir de datos externos. Se pretende mostrar cómo la concatenación directa de entradas del usuario puede derivar en ejecución de comandos no previstos y en problemas de seguridad.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
  //-- Verifica que se haya proporcionado un argumento
  if (argc != 2) {
    printf("Uso: %s <archivo>\n", argv[0]);
    return 1;
  }
  //-- Construye un comando del sistema usando la entrada del usuario
  char comando[256];
  snprintf(comando, sizeof(comando), "ls %s", argv[1]);

  //-- Ejecuta el comando en el sistema operativo
  int resultado = system(comando);

  //-- Verifica el resultado de la ejecución
  if (resultado == -1) {
    perror("Error al ejecutar el comando");
    return 1;
  }
  return 0;
}
```

Codificación recomendada

Ejecución de una funcionalidad equivalente sin invocar el intérprete de comandos del sistema. Se busca listar el contenido de un directorio usando APIs estándar del lenguaje y del sistema, evitando la ejecución indirecta de comandos y reduciendo la superficie de ataque.

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
```

```
  //-- Verifica que se haya proporcionado un argumento
  if (argc != 2) {
    printf("Uso: %s <directorio>\n", argv[0]);
    return 1;
  }
  //-- Abre el directorio indicado por el usuario
  DIR *dir = opendir(argv[1]);
  if (dir == NULL) {
    perror("No se pudo abrir el directorio");
    return 1;
  }
  //-- Lee y muestra cada entrada del directorio
  struct dirent *entry;
  while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);
  }
  //-- Cierra el directorio
  closedir(dir);

  return 0;
}
```

2.6.3.1 Descripción

La función `system()` ejecuta comandos del sistema operativo mediante el intérprete de comandos predeterminado. Su uso introduce vulnerabilidades significativas relacionadas con la **inyección de comandos** y la **ejecución de código arbitrario**.

Esta función presenta riesgos inherentes debido a que procesa la cadena de entrada a través del shell del sistema, lo que permite la interpretación de caracteres especiales y metacaracteres. Los atacantes pueden explotar esta característica para ejecutar comandos no autorizados cuando los datos de entrada no están completamente controlados.

El comportamiento de `system()` depende del entorno de ejecución y del sistema operativo, lo que introduce **inconsistencias de portabilidad** y dificulta la predicción de su comportamiento en diferentes plataformas. Además, la función hereda el entorno de ejecución del proceso padre, incluyendo variables de entorno que pueden ser manipuladas.

La mitigación principal consiste en evitar completamente el uso de `system()` y optar por alternativas más seguras. Las funciones de la familia `exec()` proporcionan mayor control sobre la ejecución de procesos externos sin involucrar al shell del sistema. Estas alternativas permiten especificar argumentos de forma individual, eliminando la interpretación de metacaracteres.

Cuando sea absolutamente necesario ejecutar comandos externos, se debe implementar una **validación estricta de entrada** mediante listas blancas de caracteres

permitidos y la sanitización de todos los datos que formarán parte del comando. La construcción dinámica de comandos debe evitarse siempre que sea posible.

Riesgos asociados

- Ejecución de comandos arbitrarios por parte de un atacante (Remote Command Execution).
- Inyección de parámetros maliciosos en la línea de comandos (Command Injection).
- Pérdida de control sobre los privilegios de ejecución.
- Acceso o modificación no autorizada de archivos y recursos del sistema.
- Exposición de información sensible a través del shell.
- Dificultad para auditar el código y validar correctamente las entradas.
- Fuga de información por errores en el manejo de la salida estándar.
- Ataques de escalada de privilegios si la aplicación corre con mayores permisos.
- Compromiso de la integridad y disponibilidad del sistema (Denegación de Servicio, DoS).
- Dependencia de la plataforma y riesgos de portabilidad.

Buenas prácticas

- Evitar la ejecución de comandos del sistema y utilizar llamadas directas a APIs del sistema o funciones estándar seguras.

```
//-- Uso de system con datos externos
char cmd[256];
snprintf(cmd, sizeof(cmd), "ls %s", user_input);
system(cmd);
```

```
//-- Uso de funciones estándar sin invocar intérprete de comandos
DIR *dir = opendir(user_input);
if (dir != NULL) {
  /* procesar directorio */
  closedir(dir);
}
```

- Reemplazar llamadas a `system` para gestión de procesos por `fork` y `exec` con validación estricta de argumentos.

```
//-- Ejecución de comando mediante system
system("cp archivo.txt /tmp/");
```

```
//-- Creación de proceso controlado con execve
pid_t pid = fork();
if (pid == 0) {
  char *args[] = {"/bin/cp", "archivo.txt", "/tmp/", NULL};
  execve("/bin/cp", args, NULL);
}
```

- Implementar funcionalidades internas en lugar de depender de comandos del sistema para reducir superficie de ataque.

```
//-- Eliminación de archivo mediante system
system("rm archivo.txt");
```

```
//-- Eliminación directa usando API del sistema
if (remove("archivo.txt") != 0) {
  /* manejar error */
}
```

2.6.3.2 Estándares metodológicos

- **MISRA C-20.11**: The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.
- **CERT/C-ENV33**: Do not call system().

2.6.4 [c.func.4] - No pasar datos inválidos a asctime

ID	c.func.4	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La función `asctime()` puede generar un **comportamiento impredecible** cuando recibe estructuras `tm` con valores fuera de los rangos válidos. Los campos de fecha y hora deben cumplir restricciones específicas para evitar desbordamientos de buffer y resultados impredecibles en la conversión a cadena de texto.

Codificación insegura

Uso de la función `asctime` para convertir una estructura `struct tm` en una cadena legible. Se muestra cómo construir manualmente la estructura de fecha y hora, y cómo una inicialización incorrecta o con valores fuera de rango puede producir resultados indefinidos o comportamientos no esperados. El objetivo es entender la importancia de validar los campos antes de llamar a la función.

```
#include <stdio.h>
```

```c
#include <time.h>

//-- Programa que construye manualmente una fecha sin validar rangos
int main(void) {
  struct tm fecha;
  char *resultado;

  //-- Inicialización manual con valores fuera de rango
  fecha.tm_year = 2025;
  fecha.tm_mon  = 15;
  fecha.tm_mday = 40;
  fecha.tm_hour = 30;
  fecha.tm_min  = 70;
  fecha.tm_sec  = 80;
  fecha.tm_wday = 0;
  fecha.tm_yday = 0;
  fecha.tm_isdst = -1;

  //-- Conversión directa sin validar los campos
  resultado = asctime(&fecha);

  //-- Impresión del resultado potencialmente incorrecto
  printf("Fecha generada: %s\n", resultado);

  return 0;
}
```

Codificación recomendada

Uso de `asctime` asegurando que la estructura `struct tm` contiene valores válidos y coherentes. Se obtiene la fecha desde el sistema y se normaliza con `localtime`, evitando manipular manualmente campos que podrían quedar fuera de rango. El objetivo es garantizar que la función reciba datos válidos y prevenir comportamientos indefinidos.

```c
#include <stdio.h>
#include <time.h>

//-- Programa que obtiene una fecha válida desde el sistema
int main(void) {
  time_t tiempo_actual;
  struct tm *fecha_valida;
  char *resultado;

  //-- Obtener tiempo actual en segundos desde Epoch
  tiempo_actual = time(NULL);
  if (tiempo_actual == (time_t)-1) {
    //-- Manejo básico de error
    printf("Error al obtener el tiempo actual\n");
```

```
    return 1;
  }
  //-- Convertir a estructura tm validada por la biblioteca
  fecha_valida = localtime(&tiempo_actual);
  if (fecha_valida == NULL) {
    //-- Manejo básico de error
    printf("Error al convertir la fecha\n");
    return 1;
  }
  //-- Conversión segura con datos válidos
  resultado = asctime(fecha_valida);

  //-- Impresión del resultado
  printf("Fecha generada: %s", resultado);

  return 0;
}
```

2.6.4.1 Descripción

La función `asctime` convierte una estructura `tm` en una cadena de caracteres que representa fecha y hora en formato legible. Esta función no valida los datos de entrada y asume que los valores de la estructura están dentro de rangos válidos.

Cuando se proporcionan valores fuera de los rangos esperados, `asctime` puede generar **desbordamientos de buffer** o producir salidas impredecibles. Los campos de la estructura `tm` deben cumplir restricciones específicas: los segundos y minutos entre 0 y 59, las horas entre 0 y 23, el día del mes entre 1 y 31, el mes entre 0 y 11, y el año como número de años desde 1900.

Los valores inválidos pueden provocar **corrupción de memoria** y comportamiento indefinido del programa. La función utiliza un buffer interno de tamaño fijo que puede ser sobrescrito si los valores generan cadenas más largas de lo esperado.

La mitigación principal consiste en **validar todos los campos** de la estructura `tm` antes de pasarla a `asctime`. Se debe verificar que cada campo esté dentro de su rango válido correspondiente. Como alternativa más segura, se recomienda utilizar `strftime`, que ofrece mayor control sobre el formato de salida y mejor manejo de casos límite.

Esta regla forma parte de las recomendaciones de seguridad que buscan prevenir vulnerabilidades relacionadas con el **manejo inseguro de funciones de tiempo** y la validación inadecuada de datos de entrada.

Riesgos asociados

- Corrupción de memoria debido a desbordamiento de buffer al recibir datos fuera de rango.

- Lectura o escritura fuera de los límites del array en `struct tm`, pudiendo resultar un comportamiento indefinido.
- Divulgación de información sensible si se leen valores no inicializados o adyacentes en memoria.
- Caída inesperada de la aplicación (crash) por acceso ilegal a memoria.
- Generación de resultados erróneos o incoherentes en la representación textual de la fecha y hora.
- Potenciales vulnerabilidades explotables para ejecución de código arbitrario si el atacante controla los datos inválidos.
- Pérdida de integridad de los datos, afectando el correcto funcionamiento o la toma de decisiones en el sistema.

Buenas prácticas

- Validar que el puntero a `struct tm` no sea `NULL` antes de llamar a `asctime`.

```
//-- uso de puntero sin validar
struct tm *info = obtenerTiempo();
char *texto = asctime(info);
```

```
//-- validación previa del puntero antes de usarlo
struct tm *info = obtenerTiempo();
if (info != NULL) {
  char *texto = asctime(info);
}
```

- Verificar que los campos de la estructura `struct tm` estén dentro de los rangos válidos antes de invocar `asctime`.

```
//-- estructura con valores fuera de rango
struct tm info;
info.tm_mon = 15;
info.tm_mday = 40;
char *texto = asctime(&info);
```

```
//-- validación de rangos antes de la llamada
struct tm info;
if (info.tm_mon >= 0 && info.tm_mon <= 11 &&
  info.tm_mday >= 1 && info.tm_mday <= 31) {
  char *texto = asctime(&info);
}
```

- Evitar pasar estructuras no inicializadas a `asctime`.

```
//-- estructura no inicializada
struct tm info;
char *texto = asctime(&info);
```

```
//-- inicialización explícita antes del uso
struct tm info = {0};
info.tm_year = 126;
info.tm_mon = 1;
info.tm_mday = 13;
char *texto = asctime(&info);
```

- Preferir alternativas seguras como `asctime_r` para evitar problemas de concurrencia.

```
//-- uso de función no reentrante en entorno concurrente
struct tm info;
char *texto = asctime(&info);
```

```
//-- uso de versión reentrante con buffer controlado
struct tm info;
char buffer[26];
asctime_r(&info, buffer);
```

2.6.4.2 Estándares metodológicos

- **CERT/C-MSC33**: Do not pass invalid data to the asctime() function.

2.7 Señales

2.7.1 [c.sig.1] - Llamar solo a funciones asincrónicamente seguras dentro de manejadores de señales

ID	c.sig.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los manejadores de señales operan en un contexto de ejecución especial donde muchas funciones estándar pueden causar **condiciones de carrera** y **estados inconsistentes**. El problema surge porque las señales pueden interrumpir la ejecución normal del programa en cualquier momento, incluso cuando otras funciones están parcialmente ejecutadas.

Codificación insegura

Gestión de señales en C donde se captura `SIGINT` (Ctrl+C) y se ejecuta lógica dentro del manejador. Se pretende mostrar qué tipo de funciones pueden invocarse desde un manejador de señales y entender el impacto que tiene usar funciones no seguras en contextos asíncronos.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void manejador(int signo) {
  //-- Manejador de señal que intenta usar funciones no asincrónicamente seguras
  printf("Señal recibida: %d\n", signo);
  //-- printf no es asincrónicamente segura y puede causar comportamiento
indefinido
  exit(0);
  //-- exit tampoco es segura en un manejador de señales
}
int main() {
  //-- Registro del manejador para la señal SIGINT
  if (signal(SIGINT, manejador) == SIG_ERR) {
    perror("Error registrando manejador");
    return EXIT_FAILURE;
  }
  //-- Bucle infinito esperando la señal
  while (1) {
    pause();
  }
  return 0;
}
```

Codificación recomendada

Gestión de señales en C donde se captura `SIGINT` (Ctrl+C) limitando las operaciones del manejador exclusivamente a funciones asincrónicamente seguras. Se busca mostrar cómo realizar una notificación simple y segura utilizando únicamente llamadas permitidas en este contexto.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

void manejador(int signo) {
  //-- Manejador que solo utiliza funciones asincrónicamente seguras
```

```
  const char mensaje[] = "Señal SIGINT recibida\n";
  //-- write es asincrónicamente segura según POSIX
  write(STDOUT_FILENO, mensaje, sizeof(mensaje) - 1);
}
int main() {
  struct sigaction sa;

  //-- Inicialización segura de la estructura sigaction
  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = manejador;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;

  //-- Registro del manejador usando sigaction
  if (sigaction(SIGINT, &sa, NULL) == -1) {
    perror("Error registrando manejador");
    return EXIT_FAILURE;
  }
  //-- Bucle principal esperando señales
  while (1) {
    pause();
  }
  return 0;
}
```

2.7.1.1 Descripción

Las funciones **async-signal-safe** son aquellas que pueden invocarse sin riesgo desde un manejador de señales porque no utilizan recursos compartidos de manera que no puedan generar condiciones de carrera o estados inconsistentes. El estándar POSIX define una lista específica de funciones que cumplen con esta característica.

Cuando se invocan funciones no seguras para señales asíncronas dentro de un manejador, se pueden producir **condiciones de carrera**, **corrupción de datos** o **bloqueos indefinidos**. Esto ocurre porque muchas funciones de biblioteca mantienen estructuras de datos internas que pueden quedar en estados inconsistentes si son interrumpidas por una señal.

Las funciones de entrada/salida estándar, las rutinas de manejo de memoria dinámica y muchas funciones de biblioteca no son async-signal-safe. Su uso dentro de manejadores de señales puede comprometer la estabilidad y predictibilidad del programa.

Para mitigar estos riesgos, se debe limitar la funcionalidad de los manejadores de señales a operaciones mínimas utilizando únicamente funciones garantizadas como async-signal-safe. Una estrategia alternativa consiste en utilizar el manejador únicamente para establecer una **bandera global** que será verificada posteriormente en

el flujo normal del programa, donde se pueden realizar operaciones más complejas de forma segura.

La implementación de mecanismos de señalización más modernos, como `signalfd` en sistemas Linux o el uso de `sigwait` para el manejo síncrono de señales, proporciona alternativas que evitan completamente las limitaciones asociadas con los manejadores de señales tradicionales.

Riesgos asociados

- Corrupción de memoria al interrumpir funciones no reentrantes.
- Condiciones de carrera y resultados impredecibles.
- Bloqueos (deadlocks) por llamadas a funciones que manipulan recursos compartidos.
- Pérdida o modificación inadvertida del estado global del programa.
- Lectura o escritura de datos inconsistentes en estructuras globales o estáticas.
- Cierre inesperado del programa (crash) debido a operaciones inseguras en el contexto del manejador.
- Escritura fuera de límites o fugas de memoria (buffer overflow o memory leak).
- Fuga de información sensible por acceso concurrente a recursos no protegidos.
- Paradigmas de sincronización rotos, dificultando el mantenimiento y detección de errores.

Buenas prácticas

- Utilizar únicamente funciones catalogadas como async-signal-safe dentro de manejadores de señales (por ejemplo, `write` en lugar de `printf`).

```
//-- Uso de printf dentro del manejador
void handler(int sig) {
  printf("Signal recibida\n");
}
```

```
//-- Uso de write que es async-signal-safe
void handler(int sig) {
  const char msg[] = "Signal recibida\n";
  write(STDOUT_FILENO, msg, sizeof(msg) - 1);
}
```

- Evitar llamadas a funciones no reentrantes (por ejemplo, `malloc`) dentro del manejador; limitarse a operaciones seguras y predecibles.

```
//-- Uso de malloc dentro del manejador
void handler(int sig) {
  char *buffer = malloc(100);
  free(buffer);
}
```

```
//-- Uso de buffer estático preasignado
static char buffer[100];
void handler(int sig) {
  buffer[0] = 'A';
}
```

- No acceder a recursos compartidos sin protección; utilizar variables de tipo `sig_atomic_t` para comunicación segura.

```
//-- Modificación de variable global no segura
int flag = 0;
void handler(int sig) {
  flag = 1;
}
```

```
//-- Uso de sig_atomic_t para acceso seguro
volatile sig_atomic_t flag = 0;
void handler(int sig) {
  flag = 1;
}
```

- Evitar funciones de E/S con buffering interno; preferir llamadas directas al sistema sin estado global compartido.

```
//-- Uso de fprintf con buffering
void handler(int sig) {
  fprintf(stderr, "Error\n");
}
```

```
//-- Uso de write sin buffering
void handler(int sig) {
  const char msg[] = "Error\n";
  write(STDERR_FILENO, msg, sizeof(msg) - 1);
}
```

2.7.1.2 Estándares metodológicos

- **CERT/C-SIG30**: Call only asynchronous-safe functions within signal handlers.

2.7.2 [c.sig.2] - No acceder a objetos compartidos dentro de manejadores de señales

ID	c.sig.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los manejadores de señales ejecutan código de forma asíncrona cuando el programa recibe una señal del sistema operativo. El acceso a **objetos compartidos** desde estos manejadores puede generar **condiciones de carrera** y **corrupción de datos**, ya que la señal puede interrumpir operaciones en curso sobre las mismas variables o estructuras de datos que el manejador intenta modificar.

Codificación insegura

Programa que maneja la señal `SIGINT` y modifica directamente una variable global compartida desde el manejador. Se busca mostrar cómo el acceso a objetos compartidos no protegidos dentro de un manejador puede generar condiciones de carrera y comportamiento indefinido.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

//-- Variable global compartida entre el flujo principal y el manejador
int contador = 0;

//-- Manejador de señal que accede y modifica un objeto compartido
void manejador(int signo) {
  //-- Incrementa el contador global
  contador++;
  //-- Imprime el valor actual (función no segura en señales)
  printf("Contador en manejador: %d\n", contador);
}
int main() {
  //-- Registro del manejador para SIGINT (Ctrl+C)
  signal(SIGINT, manejador);

  //-- Bucle principal que también usa la variable global
  while (1) {
    contador++;
    printf("Contador en main: %d\n", contador);
    sleep(1);
  }
  return 0;
}
```

Codificación recomendada

Programa que maneja la señal `SIGINT` utilizando una variable de tipo `sig_atomic_t` como bandera. El manejador solo realiza una operación segura y mínima, mientras que el acceso a recursos compartidos se realiza fuera del contexto de la señal.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

//-- Bandera segura para comunicación entre manejador y flujo principal
volatile sig_atomic_t senal_recibida = 0;

//-- Manejador de señal que solo actualiza una bandera
void manejador(int signo) {
  //-- Marca que la señal fue recibida
  senal_recibida = 1;
}
int main() {
  int contador = 0;

  //-- Registro del manejador para SIGINT (Ctrl+C)
  signal(SIGINT, manejador);

  //-- Bucle principal
  while (1) {
    //-- Verifica si la señal fue recibida
    if (senal_recibida) {
      //-- Maneja la señal fuera del contexto del manejador
      printf("SIGINT recibida. Valor contador: %d\n", contador);
      senal_recibida = 0;
    }
    contador++;
    printf("Contador en main: %d\n", contador);
    sleep(1);
  }
  return 0;
}
```

2.7.2.1 Descripción

Los manejadores de señales operan de forma asíncrona y pueden interrumpir la ejecución normal del programa en cualquier momento. El acceso a objetos compartidos desde estos manejadores genera condiciones de carrera que pueden corromper el estado del programa.

Esta restricción surge porque las señales pueden activarse durante operaciones críticas sobre variables globales, estructuras de datos o recursos del sistema. Si el manejador modifica estos objetos mientras el programa principal los está utilizando, se produce

una **inconsistencia de datos** que puede manifestarse como corrupción de memoria, valores incorrectos o comportamiento impredecible.

El problema se agrava cuando las operaciones sobre los objetos compartidos no son atómicas. Las modificaciones parciales interrumpidas por señales dejan los datos en estados intermedios inválidos, comprometiendo la integridad del programa.

Los riesgos incluyen corrupción de datos, fallos de segmentación, comportamiento indefinido y vulnerabilidades de seguridad. La naturaleza asíncrona de las señales hace que estos problemas sean difíciles de reproducir y diagnosticar durante las pruebas.

La mitigación efectiva requiere limitar las operaciones en manejadores de señales a funciones **async-signal-safe** y evitar el acceso directo a variables globales. Las técnicas incluyen el uso de variables de tipo `sig_atomic_t` para comunicación mínima, el bloqueo temporal de señales durante secciones críticas, y el diferimiento del procesamiento complejo fuera del contexto del manejador.

El cumplimiento de esta regla garantiza que los manejadores de señales no interfieran con el estado del programa principal, manteniendo la coherencia de datos y la estabilidad del sistema.

Riesgos asociados

- Condiciones de carrera que pueden derivar en corrupción de datos.
- Lectura o escritura inconsistente de variables compartidas.
- Estados inesperados en el programa difícilmente reproducibles y depurables.
- Bloqueos y deadlocks si el manejador de señal accede a recursos sincronizados.
- Violaciones de memoria (por ejemplo, acceso a memoria liberada o desreferenciada).
- Fugas de memoria o recursos al alterar incorrectamente el estado de objetos compartidos.
- Resultados impredecibles o fallas de seguridad (por ejemplo, ejecución de código inseguro).
- Dificultad para garantizar la atomicidad de las operaciones sobre objetos compartidos.
- Incremento en la superficie para ataques (potenciales exploits basados en condiciones de carrera).

Buenas prácticas

- Evitar el acceso directo a variables globales compartidas dentro de manejadores de señales; utilizar variables de tipo `volatile sig_atomic_t` para comunicar el evento.

```
//-- Modifica variable global compartida no segura
int contador = 0;
void handler(int sig) {
  contador++;
}
```

```
//-- Usa sig_atomic_t para comunicación segura
volatile sig_atomic_t flag = 0;
void handler(int sig) {
  flag = 1;
}
```

- No invocar funciones no asíncronamente seguras (como `printf`, `malloc`, `free`) desde el manejador de señales.

```
//-- Llama a función no async-signal-safe
void handler(int sig) {
  printf("Señal recibida\n");
}
```

```
//-- Solo establece indicador y procesa fuera del handler
volatile sig_atomic_t flag = 0;
void handler(int sig) {
  flag = 1;
}
void procesar() {
  if (flag) {
    printf("Señal recibida\n");
    flag = 0;
  }
}
```

- No acceder a estructuras de datos compartidas que puedan estar en estado inconsistente cuando ocurre la señal.

```
//-- Accede a estructura compartida potencialmente inconsistente
struct datos {
  int a;
  int b;
} info;
void handler(int sig) {
  info.a = 0;
  info.b = 0;
}
```

```
//-- Limita el handler a marcar estado y delega actualización
```

```
volatile sig_atomic_t reset = 0;
void handler(int sig) {
  reset = 1;
}
void actualizar() {
  if (reset) {
    info.a = 0;
    info.b = 0;
    reset = 0;
  }
}
```

- Bloquear temporalmente la señal al modificar objetos compartidos en el flujo principal para evitar condiciones de carrera.

```
//-- Modifica objeto compartido sin bloquear señal
int estado = 0;
void actualizar() {
  estado = 1;
}
```

```
//-- Bloquea señal durante sección crítica
int estado = 0;
void actualizar() {
  sigset_t set, oldset;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK, &set, &oldset);
  estado = 1;
  sigprocmask(SIG_SETMASK, &oldset, NULL);
}
```

2.7.2.2 Estándares metodológicos

- **CERT/C-SIG31**: Do not access shared objects in signal handlers.

2.7.3 [c.sig.3] - No retornar desde manejadores de señales de excepción computacional

ID	c.sig.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los manejadores de señales como `SIGFPE`, `SIGILL` o `SIGSEGV` presentan un **comportamiento inesperado** cuando retornan al punto donde se produjo la excepción.

El estándar no garantiza qué ocurrirá tras el retorno, pudiendo generar bucles infinitos, corrupción de datos o terminación inesperada del programa.

Codificación insegura

Se muestra cómo manejar una excepción computacional (`SIGFPE`) provocada por una división por cero. El objetivo es observar qué ocurre cuando el manejador de la señal simplemente imprime un mensaje y retorna, permitiendo que el flujo continúe en un estado indefinido tras la excepción.

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void manejador(int sig) {
  //-- Notifica la recepción de la señal, pero no altera el flujo del programa
  printf("Se recibió SIGFPE, retornando al flujo normal...\n");
}
int main() {
  //-- Registra el manejador para la señal SIGFPE
  signal(SIGFPE, manejador);

  //-- Provoca una división por cero para generar la excepción
  int a = 10;
  int b = 0;
  int c = a / b;

  //-- Esta línea puede ejecutarse en un estado indefinido
  printf("Resultado: %d\n", c);

  return 0;
}
```

Codificación recomendada

Se muestra cómo manejar una excepción computacional (`SIGFPE`) provocada por una división por cero, asegurando que el flujo del programa no continúe en un estado inconsistente. El objetivo es finalizar el proceso de forma controlada desde el manejador para evitar comportamiento indefinido.

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void manejador(int sig) {
  //-- Notifica la recepción de la señal y finaliza el proceso de forma segura
  printf("Se recibió SIGFPE, finalizando el programa de forma controlada...\n");
  exit(EXIT_FAILURE);
```

```
}
int main() {
  //-- Registra el manejador para la señal SIGFPE
  if (signal(SIGFPE, manejador) == SIG_ERR) {
    //-- Verifica que el registro del manejador fue exitoso
    perror("Error registrando manejador");
    return EXIT_FAILURE;
  }
  //-- Provoca una división por cero para generar la excepción
  int a = 10;
  int b = 0;
  int c = a / b;

  //-- Esta línea no debería alcanzarse si ocurre la señal
  printf("Resultado: %d\n", c);

  return EXIT_SUCCESS;
}
```

2.7.3.1 Descripción

Los manejadores de señales de excepción computacional no deben ejecutar instrucciones `return` para finalizar su ejecución. Estas señales incluyen `SIGFPE`, `SIGILL`, `SIGSEGV` y `SIGBUS`, que se generan cuando el programa encuentra errores durante la ejecución como divisiones por cero, instrucciones ilegales o violaciones de acceso a memoria.

Cuando un manejador de señal de excepción computacional retorna normalmente mediante `return`, el comportamiento del programa queda **indefinido según el estándar**. El sistema operativo no puede garantizar que la ejecución continúe de manera predecible desde el punto donde se produjo la excepción original.

La implementación típica de los sistemas operativos intenta reanudar la ejecución desde la instrucción que causó la excepción. Esto genera un **bucle infinito** donde la misma instrucción problemática se ejecuta repetidamente, provocando la señal una y otra vez. El programa queda atrapado en este ciclo sin posibilidad de recuperación.

Los riesgos asociados incluyen el consumo descontrolado de recursos del sistema, la imposibilidad de realizar tareas de limpieza necesarias, y la pérdida de control sobre el flujo de ejecución del programa. Estos problemas pueden derivar en fallos de disponibilidad y comportamientos impredecibles que comprometen la estabilidad del sistema.

La mitigación de estos riesgos requiere que los manejadores de señales de excepción computacional **terminen la ejecución del programa** de manera explícita. Las alternativas incluyen llamar a funciones de terminación del programa, realizar operaciones de limpieza antes de finalizar, o en casos específicos, modificar el contexto

de ejecución para evitar la instrucción problemática. La elección de la estrategia depende de los requisitos específicos de la aplicación y las capacidades del sistema operativo.

Riesgos asociados

- Corrupción de memoria debido a estados inconsistentes del proceso.
- Fugas de recursos por no ejecutar correctamente la limpieza (descriptores abiertos, memoria reservada, etc.).
- Pérdida de información o corrupción de datos si se omiten operaciones de finalización.
- Comportamiento indefinido al intentar ejecutar código no seguro desde el manejador.
- Condiciones de carrera si se manipulan estructuras globales fuera de contexto seguro.
- Interrupción inesperada de funciones críticas, causando fallos imprevisibles en la aplicación.
- Vulnerabilidad de seguridad por dejar el proceso en un estado explotable para atacantes.
- Difícil detección y depuración de errores intermitentes o no deterministas.

Buenas prácticas

- Finalizar el proceso de forma controlada dentro del manejador de señales de excepción en lugar de retornar al flujo normal.

```
//-- El manejador retorna y la ejecución continúa en estado indefinido
void manejador(int sig) {
  printf("Error aritmético\n");
  return;
}
```

```
//-- El manejador finaliza el proceso inmediatamente tras la excepción
void manejador(int sig) {
  _exit(1);
}
```

- Evitar continuar la ejecución tras una señal como `SIGFPE`; usar `abort()` para generar una terminación controlada con diagnóstico.

```
//-- El manejador intenta continuar después de la excepción
void manejador(int sig) {
```

```
  fprintf(stderr, "División por cero\n");
}
```

```
//-- El manejador provoca terminación inmediata y genera volcado para análisis
void manejador(int sig) {
  abort();
}
```

- No modificar el flujo normal mediante variables globales para continuar tras la excepción; terminar de forma segura.

```
int error_detectado = 0;
...
//-- El manejador marca una variable global y retorna al flujo inseguro
void manejador(int sig) {
  error_detectado = 1;
}
```

```
//-- El manejador evita retornar y termina el proceso directamente
void manejador(int sig) {
  _exit(1);
}
```

- Configurar el manejador con `sigaction` y asegurar que ante señales críticas no se reanude la ejecución normal.

```
//-- Instalación básica que permite retorno al flujo tras la señal
signal(SIGFPE, manejador);
```

```
//-- Configuración con sigaction para manejo explícito y terminación segura
struct sigaction sa;
sa.sa_handler = manejador;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGFPE, &sa, NULL);
```

2.7.3.2 Estándares metodológicos

- **CERT/C-SIG35**: Do not return from a computational exception signal handler.

2.7.4 [c.sig.4] - Asegurar que un manejador de señales sea una función simple

ID	c.sig.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los manejadores de señales que ejecutan operaciones complejas o llaman a funciones no seguras pueden generar **comportamientos impredecibles** y **condiciones de carrera**. Las señales pueden interrumpir la ejecución del programa en cualquier momento, y si el manejador modifica estructuras de datos compartidas o utiliza funciones no reentrantes, el programa puede corromperse o fallar de manera intermitente.

Codificación insegura

Gestión de una señal POSIX donde el manejador realiza múltiples operaciones como llamadas a funciones no seguras y uso de memoria dinámica. Se busca observar cómo ciertas acciones dentro del manejador pueden generar condiciones de carrera, bloqueos o comportamiento indefinido, y entender qué prácticas deben evitarse.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

//-- Manejador que realiza operaciones no seguras
void manejador(int sig) {
  //-- Uso de funciones no async-signal-safe
  printf("Señal recibida: %d\n", sig);

  //-- Uso de memoria dinámica dentro del manejador
  char *buffer = (char *)malloc(100);
  if (buffer != NULL) {
    sprintf(buffer, "Procesando señal %d", sig);
    printf("%s\n", buffer);
    free(buffer);
  }
}
int main() {
  //-- Registro del manejador de señal
  signal(SIGINT, manejador);

  //-- Bucle infinito esperando la señal
  while (1) {
    pause();
  }
  return 0;
}
```

Codificación recomendada

Gestión de una señal POSIX donde el manejador se limita a realizar una operación simple y segura, estableciendo una bandera de tipo `sig_atomic_t`. El objetivo es mostrar cómo delegar la lógica compleja al flujo principal del programa y mantener el manejador lo más simple posible para evitar comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

//-- Bandera global segura para señales
volatile sig_atomic_t senal_recibida = 0;

//-- Manejador simple y seguro
void manejador(int sig) {
  //-- Solo se actualiza una variable global atómica
  senal_recibida = 1;
}
int main() {
  //-- Registro del manejador de señal
  if (signal(SIGINT, manejador) == SIG_ERR) {
    perror("Error registrando manejador");
    return EXIT_FAILURE;
  }
  //-- Bucle principal del programa
  while (1) {
    pause();

    //-- Lógica fuera del manejador
    if (senal_recibida) {
      printf("Señal SIGINT procesada de forma segura\n");
      senal_recibida = 0;
    }
  }
  return 0;
}
```

2.7.4.1 Descripción

Los manejadores de señales deben implementarse como funciones simples que ejecuten únicamente operaciones básicas y seguras. La complejidad en estos manejadores introduce vulnerabilidades de seguridad y comportamientos impredecibles debido a las restricciones inherentes del contexto de ejecución de señales.

Las señales pueden interrumpir la ejecución del programa en cualquier momento, creando un contexto de ejecución asíncrono donde muchas funciones estándar no son seguras de usar. Solo un conjunto limitado de funciones, conocidas como **async-signal-safe**, pueden invocarse de forma segura desde un manejador de señales.

La utilización de funciones no seguras para señales puede provocar **condiciones de carrera**, **corrupción de datos** y **comportamientos indefinidos**. Estos problemas surgen cuando el manejador interrumpe operaciones que no son atómicas o que mantienen estructuras de datos en estados inconsistentes temporalmente.

Los manejadores complejos que realizan operaciones de entrada/salida, manipulación de memoria dinámica o acceso a variables globales sin sincronización adecuada incrementan significativamente el riesgo de fallos de seguridad. La naturaleza asíncrona de las señales hace que estos problemas sean difíciles de detectar y reproducir durante las pruebas.

Para mitigar estos riesgos, los manejadores deben limitarse a operaciones mínimas como establecer banderas atómicas o realizar llamadas a funciones explícitamente documentadas como async-signal-safe. La lógica compleja debe trasladarse al flujo principal del programa, donde puede ejecutarse de forma controlada y predecible.

El uso de variables de tipo `sig_atomic_t` para la comunicación entre el manejador y el programa principal proporciona un mecanismo seguro para transferir información sin introducir condiciones de carrera. Esta aproximación mantiene la simplicidad del manejador mientras preserva la funcionalidad requerida.

Riesgos asociados

- Condiciones de carrera al acceder a variables compartidas desde el manejador de señales y el flujo principal del programa.
- Interrupción de funciones no seguras para llamadas desde manejadores de señales (signal-unsafe), provocando corrupciones de memoria o comportamientos indefinidos.
- Bloqueo o deadlock si el manejador utiliza primitivas de sincronización no asíncronas.
- Pérdida de datos o corrupciones si el manejador realiza operaciones de entrada/salida no seguras.
- Caídas inesperadas del programa por realizar llamadas a funciones no reentrantes desde el manejador.
- Violaciones de seguridad debido a la ejecución de código complejo o críticas dentro del manejador, aprovechable por atacantes.
- Difícil depuración y mantenimiento del código debido a los efectos secundarios generados por manejadores no simples.

Buenas prácticas

- Limitar el manejador de señales a operaciones atómicas y seguras (async-signal-safe).

```
//-- Uso de funciones no seguras dentro del manejador
void handler(int sig) {
  printf("Signal recibida\n");
}
```

```
//-- Uso exclusivo de operaciones seguras y simples
volatile sig_atomic_t flag = 0;
void handler(int sig) {
  flag = 1;
}
```

- Evitar el uso de variables globales no atómicas dentro del manejador.

```
//-- Modificación de variable global no segura
int stop = 0;
void handler(int sig) {
  stop = 1;
}
```

```
//-- Uso de tipo sig_atomic_t para acceso seguro
volatile sig_atomic_t stop = 0;
void handler(int sig) {
  stop = 1;
}
```

- No realizar asignaciones dinámicas de memoria dentro del manejador.

```
//-- Reserva de memoria dentro del manejador
void handler(int sig) {
  char *p = malloc(100);
}
```

```
//-- Señalización simple para procesar fuera del manejador
volatile sig_atomic_t process = 0;
void handler(int sig) {
  process = 1;
}
```

- No utilizar mecanismos de sincronización complejos (mutex, semáforos) dentro del manejador.

```
//-- Uso de mutex dentro del manejador
pthread_mutex_t lock;
void handler(int sig) {
  pthread_mutex_lock(&lock);
}
```

```
//-- Señalización simple y sincronización fuera del manejador
volatile sig_atomic_t ready = 0;
void handler(int sig) {
  ready = 1;
}
```

- Registrar el manejador utilizando `sigaction` para mayor control y seguridad.

```
//-- Registro básico con signal sin configuración adicional
signal(SIGINT, handler);
```

```
//-- Registro con sigaction y estructura inicializada
struct sigaction sa;
sa.sa_handler = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGINT, &sa, NULL);
```

2.7.4.2 Estándares metodológicos

- **CERT/C-SIG30**: Call only asynchronous-safe functions within signal handlers.

2.8 Punteros

2.8.1 [c.ptr.1] - Evitar aritmética de punteros fuera del mismo array

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

La aritmética de punteros permite realizar operaciones matemáticas sobre direcciones de memoria, pero su uso fuera de los límites del array original genera **resultados impredecibles**. Cuando se incrementa, decrementa o desplaza un puntero más allá del espacio de memoria asignado al array, el programa puede acceder a **memoria no válida** o corromper datos de otras variables, causando fallos impredecibles en la ejecución.

Codificación insegura

Se muestra un programa que recorre un array de enteros usando aritmética de punteros. La intención es imprimir los elementos del array, pero se realiza un desplazamiento del puntero más allá de los límites válidos del mismo array, lo que provoca comportamiento indefinido y permite comprender el riesgo de operar fuera de los límites de memoria asignados.

```
#include <stdio.h>

int main() {
  //-- Declaración de un array de 5 enteros
  int values[5] = {10, 20, 30, 40, 50};
  int *ptr = values;

  //-- Recorrido incorrecto: se excede el tamaño del array
  for (int i = 0; i <= 5; i++) {
    //-- Se accede a memoria fuera de los límites cuando i es 5
    printf("Valor: %d\n", *(ptr + i));
  }
  return 0;
}
```

Codificación recomendada

Se presenta un programa que recorre un array de enteros utilizando aritmética de punteros de forma controlada. El objetivo es imprimir los elementos garantizando que el puntero solo se desplace dentro de los límites válidos del mismo array, evitando accesos fuera de rango y comportamiento indefinido.

```
#include <stdio.h>

int main() {
  //-- Declaración de un array de 5 enteros
  int values[5] = {10, 20, 30, 40, 50};
  int *ptr = values;
  int *end = values + 5;

  //-- Recorrido controlado dentro de los límites del mismo array
  while (ptr < end) {
    //-- Acceso seguro garantizado por comparación con el límite
    printf("Valor: %d\n", *ptr);
    ptr++;
  }
  return 0;
}
```

2.8.1.1 Descripción

La aritmética de punteros debe limitarse exclusivamente a elementos que pertenezcan al mismo array o estructura de datos contigua. Esta restricción surge de las garantías que ofrece el estándar del lenguaje respecto al comportamiento definido de las operaciones con punteros.

Cuando se realizan operaciones aritméticas entre punteros que apuntan a diferentes objetos o arrays, el comportamiento resulta indefinido según la especificación del lenguaje. El estándar únicamente garantiza que las operaciones de suma, resta y comparación entre punteros funcionen correctamente cuando ambos punteros referencian elementos del mismo array, incluyendo la posición inmediatamente posterior al último elemento válido.

Esta limitación existe porque el compilador y el sistema de gestión de memoria no pueden garantizar que diferentes objetos se encuentren en posiciones de memoria relacionadas de forma predecible. Los arrays pueden estar ubicados en segmentos de memoria distintos, separados por gaps de alineación, o incluso en diferentes regiones del espacio de direcciones virtual.

Las violaciones de esta regla pueden generar **comportamiento impredecible** que varía entre diferentes compiladores, arquitecturas y configuraciones del sistema. Los riesgos incluyen accesos a memoria no autorizada, corrupción de datos y fallos de segmentación que pueden comprometer tanto la estabilidad como la seguridad de la aplicación.

La mitigación de estos problemas requiere el uso de técnicas alternativas para la gestión de múltiples estructuras de datos. Esto incluye el empleo de índices numéricos en lugar de aritmética directa de punteros, la utilización de estructuras que encapsulen múltiples arrays, y la implementación de funciones específicas que gestionen las relaciones entre diferentes objetos de memoria de forma controlada.

Riesgos asociados

- Desbordamiento de buffer y escritura/lectura fuera de límites.
- Corrupción de memoria.
- Divulgación de información sensible.
- Condiciones de carrera difíciles de detectar.
- Ejecución de código arbitrario.
- Caída inesperada de la aplicación (crash).
- Resultados impredecibles y comportamiento indefinido.
- Introducción de vulnerabilidades explotables (por ejemplo, CVE de tipo out-of-bounds).

Buenas prácticas

- Validar que cualquier aritmética de punteros permanezca dentro de los límites del mismo array usando índices controlados.

```
//-- Avanza el puntero sin validar límites del array
int arr[5] = {0};
int *p = arr;
p = p + 7;
*p = 10;
```

```
//-- Calcula desplazamiento validando contra el tamaño del array
int arr2[5] = {0};
size_t idx = 3;
if (idx < 5) {
  arr2[idx] = 10;
}
```

- Evitar operar con punteros que referencien distintos arrays en una misma expresión aritmética.

```
//-- Resta punteros que pertenecen a diferentes arrays
int a[3] = {0};
int b[3] = {0};
ptrdiff_t diff = &b[2] - &a[0];
```

```
//-- Realiza operaciones solo entre punteros del mismo array
int c[3] = {0};
ptrdiff_t diff2 = &c[2] - &c[0];
```

- No utilizar punteros derivados después de que el array original haya dejado de ser válido.

```
//-- Retorna puntero a array local y luego realiza aritmética
int *func() {
  int local[4] = {0};
  return local + 1;
}
```

```
//-- Mantiene la aritmética dentro de un array con almacenamiento válido
int global[4] = {0};
int *func2() {
  return global + 1;
}
```

- Comprobar explícitamente los límites al iterar con punteros en lugar de confiar

en desplazamientos manuales.

```
//-- Incrementa puntero sin condición de parada ligada al array
int arr3[4] = {1,2,3,4};
int *ptr = arr3;
while (1) {
  *ptr++ = 0;
}
```

```
//-- Itera usando puntero limitado por el final del mismo array
int arr4[4] = {1,2,3,4};
int *start = arr4;
int *end = arr4 + 4;
while (start < end) {
  *start++ = 0;
}
```

2.8.1.2 Estándares metodológicos

- **MISRA C-17.1**: Pointer arithmetic shall only be applied to pointers that address an array or array element.
- **CERT/C-ARR36**: Do not subtract or compare two pointers that do not refer to the same array.
- **CERT/C-ARR37**: Do not add or subtract an integer to a pointer to a non-array object.

2.8.2 [c.ptr.2] - No usar punteros a objetos que hayan dejado de existir

ID	c.ptr.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los punteros pueden referenciar direcciones de memoria que ya no contienen objetos válidos, creando referencias colgantes que provocan un **comportamiento indefinido** al ser desreferenciados. Esta situación surge cuando se accede a memoria liberada con `free()`, variables automáticas fuera de su ámbito, o estructuras de datos temporales que han sido destruidas.

Codificación insegura

Uso de punteros que referencian memoria que deja de existir al salir de su ámbito. Se muestra cómo un puntero puede quedar apuntando a una variable local destruida al finalizar una función, generando comportamiento indefinido. El objetivo es entender el

riesgo de retornar direcciones de variables automáticas y cómo esto afecta la estabilidad y seguridad del programa.

```
#include <stdio.h>

//-- Función que retorna la dirección de una variable local
int* obtenerDireccion() {
  //-- Variable local en la pila
  int valor = 42;

  //-- Se retorna la dirección de una variable que dejará de existir
  return &valor;
}
int main() {
  //-- Se recibe un puntero a memoria inválida
  int *ptr = obtenerDireccion();

  //-- Uso de un puntero colgante (comportamiento indefinido)
  printf("Valor apuntado: %d\n", *ptr);

  return 0;
}
```

Codificación recomendada

Gestión correcta del tiempo de vida de los objetos apuntados. Se evita retornar la dirección de variables locales y se garantiza que la memoria referenciada siga siendo válida después de salir de la función, utilizando memoria dinámica y liberándola de forma controlada. El objetivo es asegurar que ningún puntero apunte a objetos que ya no existen.

```
#include <stdio.h>
#include <stdlib.h>

//-- Función que reserva memoria dinámica y retorna un puntero válido
int* obtenerDireccion() {
  //-- Reserva de memoria en el heap
  int *valor = (int*)malloc(sizeof(int));

  //-- Verificación de reserva exitosa
  if (valor == NULL) {
    return NULL;
  }
  //-- Inicialización del valor
  *valor = 42;

  //-- Se retorna un puntero a memoria válida
  return valor;
```

```
}
int main() {
  //-- Recepción del puntero válido
  int *ptr = obtenerDireccion();

  //-- Verificación antes de uso
  if (ptr == NULL) {
    return 1;
  }
  //-- Uso seguro del puntero
  printf("Valor apuntado: %d\n", *ptr);

  //-- Liberación de memoria para evitar fugas
  free(ptr);

  return 0;
}
```

2.8.2.1 Descripción

Los punteros que referencian objetos que han dejado de existir en memoria constituyen una fuente significativa de vulnerabilidades y comportamientos indefinidos. Esta situación se produce cuando un puntero mantiene una dirección de memoria que anteriormente contenía un objeto válido, pero dicho objeto ha sido liberado, ha salido de su ámbito de vida o ha sido destruido por cualquier otro mecanismo.

El problema surge porque el sistema de gestión de memoria no invalida automáticamente los punteros cuando los objetos a los que apuntan dejan de existir. El puntero conserva la dirección de memoria original, pero el contenido de esa ubicación puede haber sido reutilizado para otros propósitos, sobrescrito con datos diferentes o marcado como no válido por el sistema operativo.

Los riesgos asociados incluyen corrupción de datos, ejecución de código malicioso, fallos de segmentación y comportamientos impredecibles del programa. Estos problemas pueden manifestarse de forma intermitente, lo que dificulta su detección durante las fases de desarrollo y pruebas.

La mitigación de este problema requiere un control riguroso del ciclo de vida de los punteros y los objetos que referencian. Las estrategias incluyen la invalidación explícita de punteros tras la liberación de memoria, el uso de técnicas de programación que limiten el ámbito de vida de los punteros, y la implementación de mecanismos de verificación que detecten accesos a memoria no válida.

Las herramientas de análisis estático y dinámico pueden identificar muchos casos de uso de punteros inválidos, especialmente cuando se combinan con técnicas de instrumentación del código que monitorizan los accesos a memoria durante la ejecución del programa.

Riesgos asociados

- Dereferenciación de punteros colgantes, que conduce a comportamientos indefinidos.
- Lectura o escritura de memoria liberada, posibilitando corrupción de datos.
- Vulnerabilidades de seguridad como violaciones de acceso a memoria (segmentation fault).
- Fugas de información potencial, permitiendo a un atacante leer datos sensibles previamente almacenados.
- Exposición a ataques de uso después de liberación (use-after-free), vectores comunes en explotación de vulnerabilidades.
- Inestabilidad y fallos inesperados del programa en tiempo de ejecución.
- Dificultad en la detección y depuración de errores, incrementando el coste y tiempo de desarrollo.
- Condiciones de carrera cuando múltiples hilos acceden a memoria reutilizada.

Buenas prácticas

- Evitar devolver punteros a variables locales definidas en la pila.

```
//-- Uso de variable local cuyo tiempo de vida termina al salir de la función
int* obtenerValor() {
  int x = 10;
  return &x;
}
```

```
//-- Uso de memoria dinámica o almacenamiento estático para extender el tiempo de
vida
int* obtenerValor() {
  int* x = malloc(sizeof(int));
  if (x != NULL) {
    *x = 10;
  }
  return x;
}
```

- No utilizar punteros después de liberar la memoria asociada.

```
//-- Liberación de memoria y uso posterior del puntero
int* ptr = malloc(sizeof(int));
*ptr = 5;
free(ptr);
printf("%d", *ptr);
```

```
//-- Liberación controlada y anulación del puntero tras su uso
int* ptr = malloc(sizeof(int));
if (ptr != NULL) {
  *ptr = 5;
  printf("%d", *ptr);
  free(ptr);
  ptr = NULL;
}
```

- No almacenar direcciones de variables locales en estructuras con mayor tiempo de vida.

```
//-- Asignación de dirección de variable local a estructura persistente
typedef struct {
  int* dato;
} Contenedor;

Contenedor c;
void asignar() {
  int valor = 20;
  c.dato = &valor;
}
```

```
//-- Reserva dinámica para asegurar que el objeto persista
typedef struct {
  int* dato;
} Contenedor;

Contenedor c;
void asignar() {
  c.dato = malloc(sizeof(int));
  if (c.dato != NULL) {
    *c.dato = 20;
  }
}
```

- Limitar el ámbito de los punteros y evitar exponer referencias internas a datos temporales.

```
//-- Retorno de puntero a buffer local
char* obtenerCadena() {
  char buffer[16] = "hola";
  return buffer;
}
```

```
//-- Uso de buffer proporcionado por el llamador para mantener control del ciclo de vida
```

```
void obtenerCadena(char* buffer, size_t tam) {
  if (tam >= 5) {
    strncpy(buffer, "hola", tam - 1);
    buffer[tam - 1] = '\0';
  }
}
```

2.8.2.2 Estándares metodológicos

- **MISRA C-17.2**: Pointer subtraction shall only be applied to pointers that address elements of the same array.

2.8.3 [c.ptr.3] - No acceder a memoria fuera de los límites de un objeto

ID	c.ptr.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los accesos a memoria fuera de los límites de arrays, buffers y otros objetos constituyen una de las vulnerabilidades más frecuentes y peligrosas. Estos errores pueden provocar **corrupción de datos**, **ejecución de código arbitrario** o **fallos del sistema**. Las operaciones de lectura y escritura que exceden los límites asignados generan comportamiento indefinido y representan vectores de ataque comunes en exploits de seguridad.

Codificación insegura

Se muestra un programa que trabaja con un array de tamaño fijo y accede a sus elementos mediante un índice. La intención es observar cómo un acceso sin validación puede provocar lecturas o escrituras fuera de los límites del array, generando comportamiento indefinido y posibles fallos de seguridad.

```
#include <stdio.h>
#include <stdlib.h>

//-- Punto de entrada del programa
int main(void) {
  //-- Declaración de un array de 5 enteros
  int numbers[5] = {10, 20, 30, 40, 50};

  //-- Índice fuera de los límites válidos (0 a 4)
  int index = 7;

  //-- Acceso sin validación al array
  printf("Valor en la posicion %d: %d\n", index, numbers[index]);
```

```
  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Se muestra un programa que accede a un array de tamaño fijo validando previamente que el índice esté dentro de los límites permitidos. El objetivo es aprender a prevenir accesos fuera de rango mediante comprobaciones explícitas antes de leer o escribir en memoria.

```
#include <stdio.h>
#include <stdlib.h>

//-- Punto de entrada del programa
int main(void) {
  //-- Declaración de un array de 5 enteros
  int numbers[5] = {10, 20, 30, 40, 50};

  //-- Índice proporcionado para el acceso
  int index = 3;
  int size = sizeof(numbers) / sizeof(numbers[0]);

  //-- Validación de límites antes de acceder al array
  if (index >= 0 && index < size) {
    //-- Acceso seguro dentro de los límites
    printf("Valor en la posicion %d: %d\n", index, numbers[index]);
  } else {
    //-- Manejo de error si el índice es inválido
    fprintf(stderr, "Error: indice fuera de rango\n");
    return EXIT_FAILURE;
  }
  //-- Fin del programa
  return EXIT_SUCCESS;
}
```

2.8.3.1 Descripción

Los accesos a memoria fuera de los límites de un objeto constituyen una de las vulnerabilidades más frecuentes y peligrosas en el desarrollo de software. Esta regla establece que todas las operaciones de lectura y escritura deben mantenerse dentro del espacio de memoria asignado al objeto correspondiente.

Un objeto define un área contigua de memoria con límites específicos determinados por su tipo y tamaño. Los accesos válidos deben respetar estos límites, desde la primera posición hasta la última posición válida del objeto. Cualquier operación que intente acceder a posiciones anteriores al inicio o posteriores al final del objeto viola esta regla.

Los desbordamientos de buffer representan el caso más común de esta violación, especialmente en operaciones con arrays, cadenas de caracteres y estructuras de datos. Las funciones de manipulación de memoria y cadenas requieren especial atención, ya que, frecuentemente, operan con punteros y tamaños que pueden exceder los límites del objeto destino.

La aritmética de punteros introduce riesgos adicionales cuando los cálculos resultan en direcciones fuera del rango válido del objeto. Los punteros que apuntan más allá del final del objeto o antes de su inicio pueden provocar comportamientos indefinidos al ser desreferenciados.

Las consecuencias de violar esta regla incluyen corrupción de datos adyacentes, ejecución de código arbitrario, fallos de segmentación y comportamientos impredecibles del programa. Estos problemas pueden manifestarse de forma inmediata o permanecer latentes hasta que se den condiciones específicas de ejecución.

La mitigación efectiva requiere validación rigurosa de índices y tamaños antes de realizar operaciones de memoria. Las herramientas de análisis estático y dinámico pueden detectar muchos casos de accesos fuera de límites durante el desarrollo y las pruebas. El uso de funciones seguras que incluyen verificación de límites proporciona una capa adicional de protección.

Riesgos asociados

- Corrupción de memoria y datos no intencionada.
- Lectura o escritura de datos confidenciales.
- Ejecución de código arbitrario (exploits de buffer overflow).
- Condiciones de carrera al sobrescribir memoria compartida.
- Comportamiento indefinido del programa (fallos y cuelgues).
- Fallas de segmentación (segmentation faults) y denegación de servicio.
- Pérdida de integridad y consistencia en las estructuras de datos.
- Vulnerabilidad frente a ataques de seguridad como remote code execution.
- Dificultad en el rastreo y depuración de errores.

Buenas prácticas

- Validar siempre los límites de los índices antes de acceder a arrays.

```
//-- Acceso sin validar límites del índice
int arr[10];
int i = 10;
arr[i] = 5;
```

```
//-- Validación previa del índice dentro del rango válido
int arr2[10];
int j = 9;
if (j >= 0 && j < 10) {
  arr2[j] = 5;
}
```

- Utilizar `sizeof` para calcular el número real de elementos de un array.

```
//-- Uso de límite fijo incorrecto que puede exceder el tamaño real
int data[5];
for (int i = 0; i <= 5; i++) {
  data[i] = i;
}
```

```
//-- Cálculo dinámico del número de elementos con sizeof
int data2[5];
size_t n = sizeof(data2) / sizeof(data2[0]);
for (size_t i = 0; i < n; i++) {
  data2[i] = i;
}
```

- Comprobar el tamaño del buffer antes de copiar datos en memoria.

```
//-- Copia sin verificar el tamaño del destino
char src[] = "Texto largo";
char dest[5];
strcpy(dest, src);
```

```
//-- Copia limitada al tamaño del buffer destino
char src2[] = "Texto largo";
char dest2[5];
strncpy(dest2, src2, sizeof(dest2) - 1);
dest2[sizeof(dest2) - 1] = '\0';
```

- Validar el tamaño de memoria asignada dinámicamente antes de acceder a ella.

```
//-- Acceso fuera del tamaño reservado dinámicamente
int *ptr = malloc(3 * sizeof(int));
ptr[3] = 10;
```

```
//-- Acceso limitado al rango reservado dinámicamente
int *ptr2 = malloc(3 * sizeof(int));
if (ptr2 != NULL) {
  for (size_t i = 0; i < 3; i++) {
    ptr2[i] = 0;
```

```
  }
}
```

- Evitar la aritmética de punteros que exceda los límites del objeto.

```
//-- Incremento de puntero más allá del objeto
int nums[4] = {0};
int *p = nums;
p = p + 5;
*p = 1;
```

```
//-- Control del desplazamiento dentro de los límites válidos
int nums2[4] = {0};
int *p2 = nums2;
size_t offset = 3;
if (offset < 4) {
  *(p2 + offset) = 1;
}
```

2.8.3.2 Estándares metodológicos

- **MISRA C-17.4**: Array indexing shall be the only allowed form of pointer arithmetic.
- **CERT/C-ARR30**: Do not form or use out-of-bounds pointers or array subscripts.

2.8.4 [c.ptr.4] - No almacenar punteros devueltos por funciones que los invalidan

ID	c.ptr.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de biblioteca que devuelven punteros a memoria interna pueden **invalidar estos punteros** en llamadas posteriores. Almacenar y reutilizar estos punteros tras nuevas invocaciones genera un **comportamiento indefinido**, ya que la memoria referenciada puede haber sido modificada o liberada por la implementación de la función.

Codificación insegura

Uso de `strtok` para dividir cadenas y almacenamiento de los punteros devueltos para utilizarlos después de nuevas llamadas a la misma función. Se busca mostrar cómo el estado interno de la función puede invalidar esos punteros y provocar comportamiento indefinido.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  //-- Declaración de dos cadenas independientes
  char cadena1[] = "usuario:admin";
  char cadena2[] = "rol:root";

  //-- Se obtiene el primer token de la primera cadena
  char *token_guardado = strtok(cadena1, ":");

  //-- Nueva llamada a strtok con otra cadena que invalida el estado interno
  strtok(cadena2, ":");

  //-- Uso del puntero previamente almacenado
  printf("Token almacenado: %s\n", token_guardado);

  return 0;
}
```

Codificación recomendada

Uso de `strtok` copiando el contenido del token a un buffer propio antes de realizar nuevas llamadas que puedan modificar el estado interno. Se pretende aprender a evitar el almacenamiento de punteros que puedan quedar inválidos y a trabajar con copias seguras de los datos.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  //-- Declaración de dos cadenas independientes
  char cadena1[] = "usuario:admin";
  char cadena2[] = "rol:root";

  //-- Buffer propio para almacenar copia segura del token
  char copia_token[50];

  //-- Se obtiene el primer token de la primera cadena
  char *token = strtok(cadena1, ":");

  //-- Copia del token a memoria propia antes de otra llamada a strtok
  if (token != NULL) {
    strncpy(copia_token, token, sizeof(copia_token) - 1);
    copia_token[sizeof(copia_token) - 1] = '\0';
  }

  //-- Nueva llamada a strtok que ya no afecta a la copia realizada
```

```
    strtok(cadena2, ":");

    //-- Uso seguro del contenido copiado
    printf("Token almacenado de forma segura: %s\n", copia_token);

    return 0;
}
```

2.8.4.1 Descripción

Funciones como `strtok`, `asctime`, `ctime`, `gmtime`, `localtime`, `getenv`, `setlocale` y `tmpnam` mantienen buffers internos estáticos que reutilizan entre llamadas. Cada nueva invocación puede sobrescribir el contenido previo, invalidando cualquier puntero obtenido anteriormente.

El problema se agrava en entornos multihilo, donde diferentes hilos pueden invocar estas funciones simultáneamente, creando condiciones de carrera que comprometen la integridad de los datos. La naturaleza estática del almacenamiento interno hace que el comportamiento sea impredecible cuando múltiples partes del programa acceden concurrentemente a estos recursos compartidos.

Los riesgos asociados incluyen **corrupción silenciosa de datos**, donde los valores cambian sin indicación explícita, y **accesos a memoria inconsistente** que pueden manifestarse como fallos intermitentes difíciles de diagnosticar. En sistemas críticos, estos problemas pueden escalar a vulnerabilidades de seguridad explotables.

La mitigación efectiva requiere **copia inmediata** de los datos apuntados a buffers controlados por el programa, uso de **alternativas thread-safe** cuando estén disponibles, e implementación de **sincronización adecuada** en entornos concurrentes. Las versiones reentrantes de estas funciones, identificadas típicamente con el sufijo `_r`, proporcionan interfaces más seguras al requerir buffers proporcionados por el usuario.

Riesgos asociados

- Desreferenciación de punteros colgantes (dangling pointers), lo que puede provocar corrupción de memoria o fallos de segmentación (segfault).
- Lectura o escritura en áreas de memoria liberadas, resultando en comportamiento indefinido de la aplicación.
- Exposición de datos sensibles residuales en memoria reutilizada por otras partes del programa.
- Introducción de vulnerabilidades de seguridad explotables, como Use-After-Free.
- Condiciones de carrera al trabajar con memoria compartida y punteros invalidados.

- Dificultades en la depuración por errores intermitentes y difíciles de reproducir.
- Posibilidad de fuga de memoria indirecta, si los punteros inválidos impiden la liberación adecuada de recursos.

Buenas prácticas

- Evitar almacenar punteros devueltos por funciones que retornan memoria interna reutilizable (por ejemplo, buffers estáticos).

```
//-- Uso de puntero a buffer interno reutilizado
char *p1 = strtok(cadena, ",");
char *p2 = strtok(NULL, ",");
printf("%s", p1);
```

```
//-- Copia inmediata del contenido antes de nueva llamada
char *tmp = strtok(cadena, ",");
char copia[50];
strncpy(copia, tmp, sizeof(copia) - 1);
copia[sizeof(copia) - 1] = '\0';
char *p2b = strtok(NULL, ",");
printf("%s", copia);
```

- No conservar punteros a memoria liberada por funciones que invalidan recursos dinámicos.

```
//-- Uso de puntero después de liberar memoria
char *buffer = malloc(100);
free(buffer);
buffer[0] = 'A';
```

```
//-- Invalidación explícita del puntero tras liberar
char *buffer2 = malloc(100);
free(buffer2);
buffer2 = NULL;
if (buffer2 != NULL) {
  buffer2[0] = 'A';
}
```

- No almacenar direcciones de variables locales devueltas por funciones.

```
//-- Retorno de dirección de variable local
char* obtenerMensaje() {
  char msg[20] = "Hola";
  return msg;
}
```

```
//-- Uso de memoria dinámica o parámetro de salida
char* obtenerMensajeSeguro() {
  char *msg = malloc(20);
  strcpy(msg, "Hola");
  return msg;
}
```

2.8.4.2 Estándares metodológicos

- **CERT/C-ENV34**: Do not store pointers returned by certain functions.

2.8.5 [c.ptr.5] - No desreferenciar punteros nulos

ID	**c.ptr.5**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los punteros nulos representan uno de los errores más frecuentes en programación, causando **fallos de segmentación** y terminación inesperada de programas. La desreferenciación de un puntero `NULL` produce estados indefinidos que pueden ser explotados por atacantes para ejecutar código malicioso o provocar denegación de servicio.

Codificación insegura

Acceso a memoria mediante un puntero que puede tomar el valor `NULL` sin realizar validaciones previas. Se muestra cómo una asignación directa a `NULL` seguida de su desreferenciación provoca un comportamiento indefinido y normalmente una falla en tiempo de ejecución. El objetivo es observar el riesgo de no validar punteros antes de usarlos.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Declaración de un puntero a entero
  int *ptr = NULL;

  //-- Simulación de una condición donde el puntero no apunta a memoria válida
  printf("Intentando escribir en un puntero NULL...\n");

  //-- Desreferenciación directa sin validación previa (comportamiento indefinido)
  *ptr = 10;

  //-- Esta línea probablemente no se ejecutará debido a una falla previa
  printf("Valor asignado: %d\n", *ptr);
```

```
  return 0;
}
```

Codificación recomendada

Uso seguro de punteros verificando explícitamente que no sean `NULL` antes de desreferenciarlos. Se reserva memoria dinámicamente y se valida tanto el resultado de la reserva como el puntero antes de acceder a la memoria, evitando comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Declaración de un puntero a entero
  int *ptr = NULL;

  //-- Reserva dinámica de memoria para un entero
  ptr = (int *)malloc(sizeof(int));

  //-- Verificación de que la memoria fue asignada correctamente
  if (ptr == NULL) {
    //-- Manejo de error si malloc falla
    fprintf(stderr, "Error: no se pudo asignar memoria.\n");
    return 1;
  }
  //-- Validación adicional antes de desreferenciar el puntero
  if (ptr != NULL) {
    *ptr = 10;
    printf("Valor asignado: %d\n", *ptr);
  }
  //-- Liberación de la memoria reservada
  free(ptr);
  ptr = NULL;

  return 0;
}
```

2.8.5.1 Descripción

Los punteros nulos representan direcciones de memoria inválidas que no apuntan a ningún objeto válido. Desreferenciar estos punteros mediante operadores como `*`, `->` o acceso a arrays constituye un comportamiento indefinido que puede provocar fallos de segmentación, terminación abrupta del programa o corrupción de datos.

Esta regla establece que **no se debe desreferenciar un puntero sin verificar previamente que no sea nulo**. El problema surge frecuentemente cuando se asume incorrectamente que una función de asignación de memoria ha sido exitosa, cuando se reciben punteros como parámetros sin validación, o cuando se utilizan punteros después de haber sido liberados y establecidos a `NULL`.

Los riesgos asociados incluyen **vulnerabilidades de seguridad** que pueden ser explotadas por atacantes para ejecutar código malicioso, además de inestabilidad del sistema y pérdida de datos. En sistemas críticos, estos fallos pueden tener consecuencias graves para la disponibilidad e integridad del servicio.

La mitigación efectiva requiere implementar **validaciones sistemáticas** de punteros antes de su uso, establecer políticas claras para el manejo de valores de retorno de funciones de asignación de memoria, y adoptar patrones de programación defensiva que asuman la posibilidad de punteros nulos en todas las interfaces de función.

Las herramientas de análisis estático pueden detectar muchos casos de desreferenciación potencial de punteros nulos durante la compilación, mientras que las herramientas de análisis dinámico y depuración pueden identificar estos problemas durante la ejecución del programa.

Riesgos asociados

- Producción de fallos de segmentación (segmentation fault), causando terminación inesperada de la aplicación.
- Generación de condiciones de denegación de servicio (DoS) al provocar la caída del programa.
- Dificultad en el aislamiento y la corrección de errores debido a rastros poco claros en tiempo de ejecución.
- Potencial para explotar vulnerabilidades de seguridad a través de ataques dirigidos a la gestión indebida de punteros.
- Compromiso de la integridad y disponibilidad del sistema o aplicación afectada.
- Comportamiento indefinido que podría conducir a resultados erróneos o corrupción de datos.
- Exposición a vulnerabilidades específicas de ciertos sistemas operativos, que podrían aprovecharse para la ejecución de código no autorizado.

Buenas prácticas

- Validar explícitamente los punteros antes de desreferenciarlos.

```
//-- Desreferenciar sin validar el puntero
int value = *ptr;
```

```
//-- Verificar puntero antes de usarlo
if (ptr == NULL) {
  return;
}
int value = *ptr;
```

- Comprobar el resultado de funciones que pueden devolver `NULL` antes de utilizar el recurso.

```
//-- Usar memoria reservada sin comprobar resultado
char *buffer = malloc(100);
buffer[0] = 'A';
```

```
//-- Validar resultado de malloc antes de acceder
char *buffer = malloc(100);
if (buffer == NULL) {
  return;
}
buffer[0] = 'A';
```

- Asignar `NULL` después de liberar memoria y validar antes de reutilizar el puntero.

```
//-- Liberar y reutilizar puntero sin validación
free(data);
data[0] = 1;
```

```
//-- Asignar NULL y comprobar antes de usar
free(data);
data = NULL;
if (data != NULL) {
  data[0] = 1;
}
```

2.8.5.2 Estándares metodológicos

- **CERT/C-EXP34**: Do not dereference null pointers.

2.8.6 [c.ptr.6] - No provocar comportamiento indefinido al usar punteros restrict

ID	c.ptr.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los punteros `restrict` permiten al compilador realizar optimizaciones agresivas al asumir que no existe **aliasing** entre las regiones de memoria referenciadas. El uso incorrecto de esta cualificación puede causar un **comportamiento indefinido** cuando múltiples punteros acceden a las mismas ubicaciones de memoria, invalidando las suposiciones del compilador y produciendo resultados impredecibles en el programa.

Codificación insegura

Uso de punteros `restrict` en una función que opera sobre dos arrays. Se busca mostrar cómo el compilador asume que los punteros no apuntan a la misma región de memoria y qué ocurre cuando esa promesa no se cumple, generando comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>

//-- Suma el contenido de src en dst asumiendo que no hay solapamiento
void sumar(int *restrict dst, int *restrict src, size_t n) {
  size_t i;
  for (i = 0; i < n; i++) {
    dst[i] += src[i];
  }
}
int main(void) {
  int datos[3] = {1, 2, 3};

  //-- Se pasan punteros que apuntan al mismo array
  //-- Viola la promesa de restrict y provoca comportamiento indefinido
  sumar(datos, datos, 3);

  size_t i;
  for (i = 0; i < 3; i++) {
    printf("%d\n", datos[i]);
  }
  return 0;
}
```

Codificación recomendada

Uso correcto de punteros `restrict` garantizando que las regiones de memoria no se solapan. Se respeta el contrato que impone `restrict`, permitiendo optimizaciones seguras por parte del compilador.

```
#include <stdio.h>
#include <stdlib.h>

//-- Suma el contenido de src en dst asumiendo que no hay solapamiento
void sumar(int *restrict dst, int *restrict src, size_t n) {
  size_t i;
  for (i = 0; i < n; i++) {
    dst[i] += src[i];
  }
}
int main(void) {
  int destino[3] = {1, 2, 3};
  int origen[3] = {4, 5, 6};

  //-- Se garantiza que destino y origen no comparten memoria
  //-- Se cumple el contrato de restrict evitando comportamiento indefinido
  sumar(destino, origen, 3);

  size_t i;
  for (i = 0; i < 3; i++) {
    printf("%d\n", destino[i]);
  }
  return 0;
}
```

2.8.6.1 Descripción

Los punteros `restrict` constituyen una optimización del compilador que permite asumir que el objeto apuntado solo es accesible a través de ese puntero específico durante su tiempo de vida. Esta restricción permite al compilador generar código más eficiente al eliminar verificaciones redundantes de aliasing.

El **comportamiento indefinido** surge cuando se viola la garantía de acceso exclusivo que implica la palabra clave `restrict`. Esto ocurre cuando múltiples punteros acceden al mismo objeto de memoria y al menos uno de ellos está marcado como `restrict`, o cuando se accede al objeto a través de otros medios mientras existe un puntero `restrict` activo.

Las violaciones de las restricciones de aliasing pueden manifestarse de forma impredecible. El compilador optimiza el código asumiendo que no existe solapamiento entre los objetos apuntados, lo que puede resultar en **corrupción silenciosa de datos**,

cálculos incorrectos o comportamientos erráticos que varían según la configuración del compilador y las optimizaciones aplicadas.

La detección de estas violaciones presenta dificultades significativas durante el desarrollo, ya que el código puede funcionar correctamente en configuraciones de depuración, pero fallar en versiones optimizadas. Los errores pueden permanecer latentes hasta que cambios en el compilador o en las opciones de optimización expongan el comportamiento indefinido.

La mitigación efectiva requiere un **análisis cuidadoso de los patrones de acceso a memoria** antes de aplicar la calificación `restrict`. Es necesario verificar que no existan alias directos o indirectos al objeto durante el período de vigencia del puntero restringido. Las herramientas de análisis estático pueden ayudar a identificar violaciones potenciales, aunque su detección completa requiere análisis manual del flujo de datos.

El uso conservador de `restrict` limita su aplicación a casos donde se puede garantizar matemáticamente la ausencia de aliasing. Esto incluye la documentación explícita de las precondiciones de las funciones que utilizan parámetros `restrict` y la implementación de verificaciones en tiempo de ejecución cuando sea factible.

Riesgos asociados

- Acceso a memoria no válida, que puede causar fallos de segmentación (segfaults).
- Corrupción de datos en memoria por modificaciones inesperadas de variables.
- Resultados impredecibles en la ejecución del programa.
- Vulnerabilidades de seguridad, tales como desbordamientos de buffer o exposición de información sensible.
- Dificultad para depurar y mantener el código debido al comportamiento no determinista.
- Pérdida de portabilidad, ya que el código puede comportarse diferente según el compilador o la arquitectura.
- Posibles errores de optimización por parte del compilador, que asume la validez de las restricciones impuestas por restrict.
- Ataques de tipo “use-after-free” si se manipulan áreas de memoria ya liberadas.

Buenas prácticas

- Evitar que dos punteros `restrict` apunten a regiones de memoria solapadas dentro del mismo ámbito.

```
//-- Uso de punteros restrict que referencian la misma memoria
void suma(int *restrict a, int *restrict b) {
```

```
  *a = *a + *b;
}
int x = 5;
suma(&x, &x);
```

```
//-- Uso de punteros restrict con regiones de memoria distintas
void suma_segura(int *restrict a, int *restrict b) {
  *a = *a + *b;
}
int x = 5;
int y = 3;
suma_segura(&x, &y);
```

- No derivar punteros adicionales que accedan a la misma región apuntada por un `restrict` dentro del mismo bloque.

```
//-- Creación de alias adicional sobre puntero restrict
void actualizar(int *restrict p) {
  int *q = p;
  *p = 10;
  *q = 20;
}
```

```
//-- Uso exclusivo del puntero restrict sin alias adicionales
void actualizar_seguro(int *restrict p) {
  *p = 10;
  *p = 20;
}
```

- Evitar pasar a una función con parámetros `restrict` punteros derivados de una misma estructura o array que puedan solaparse.

```
//-- Paso de elementos solapados del mismo array a parámetros restrict
void procesar(int *restrict a, int *restrict b) {
  *a += *b;
}
int datos[2] = {1, 2};
procesar(&datos[0], &datos[0]);
```

```
//-- Paso de elementos independientes a parámetros restrict
void procesar_seguro(int *restrict a, int *restrict b) {
  *a += *b;
}
int x = 1;
int y = 2;
```

```
procesar_seguro(&x, &y);
```

2.8.6.2 Estándares metodológicos

- **CERT/C-EXP43**: Avoid undefined behavior when using restrict-qualified pointers.

2.9 Gestión de memoria

2.9.1 [c.mem.1] - No acceder a memoria ya liberada

ID	c.mem.1	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El acceso a memoria previamente liberada mediante `free()` constituye un error crítico en el programa. Esta situación, conocida como **use-after-free**, ocurre cuando se intenta leer o escribir en direcciones de memoria que ya no pertenecen al programa, pudiendo causar corrupción de datos, fallos de segmentación o vulnerabilidades de seguridad explotables por atacantes.

Codificación insegura

Gestión dinámica de memoria donde se reserva un bloque, se libera y, posteriormente, se intenta seguir utilizando el puntero. El objetivo es mostrar qué ocurre cuando se accede a memoria ya liberada y cómo esto puede provocar comportamiento indefinido, fallos o vulnerabilidades.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  //-- Reserva memoria para un entero
  int *ptr = (int *)malloc(sizeof(int));
  if (ptr == NULL) {
    printf("Error al asignar memoria\n");
    return 1;
  }
  //-- Asigna un valor al entero reservado
  *ptr = 42;
  printf("Valor antes de liberar: %d\n", *ptr);

  //-- Libera la memoria reservada
  free(ptr);

  //-- Accede a memoria ya liberada (comportamiento indefinido)
```

```
  printf("Valor después de liberar: %d\n", *ptr);

  return 0;
}
```

Codificación recomendada

Gestión dinámica de memoria donde se reserva un bloque, se libera y se evita cualquier acceso posterior al mismo. Se muestra cómo invalidar el puntero tras liberar la memoria y cómo comprobar su estado antes de usarlo para prevenir accesos indebidos.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  //-- Reserva memoria para un entero
  int *ptr = (int *)malloc(sizeof(int));
  if (ptr == NULL) {
    printf("Error al asignar memoria\n");
    return 1;
  }
  //-- Asigna un valor al entero reservado
  *ptr = 42;
  printf("Valor antes de liberar: %d\n", *ptr);

  //-- Libera la memoria y anula el puntero
  free(ptr);
  ptr = NULL;

  //-- Verifica que el puntero es NULL antes de usarlo
  if (ptr != NULL) {
    printf("Valor después de liberar: %d\n", *ptr);
  } else {
    printf("El puntero ya no apunta a memoria válida\n");
  }
  return 0;
}
```

2.9.1.1 Descripción

Esta regla prohíbe el acceso a memoria que ha sido previamente liberada mediante funciones como `free()` o `realloc()`. Una vez que se libera un bloque de memoria, cualquier intento de lectura o escritura sobre esa dirección constituye un comportamiento indefinido según el estándar.

El problema surge porque los punteros no se invalidan automáticamente tras la liberación de memoria. El puntero mantiene la misma dirección, pero el contenido de

esa ubicación ya no pertenece al programa. El sistema operativo o el gestor de memoria pueden reutilizar ese espacio para otros propósitos, modificar su contenido o marcarlo como inaccesible.

Los riesgos asociados incluyen **corrupción de datos**, ya que la memoria liberada puede contener información de otras partes del programa. También pueden producirse **fallos de segmentación** cuando el sistema detecta el acceso ilegal, o peor aún, el programa puede continuar ejecutándose con datos incorrectos, generando resultados impredecibles.

Desde una perspectiva de seguridad, esta vulnerabilidad puede ser explotada por atacantes para ejecutar código malicioso o acceder a información sensible que permanezca en la memoria liberada.

La mitigación se basa en el control riguroso del ciclo de vida de los punteros. Esto incluye establecer punteros a `NULL` inmediatamente después de liberar la memoria, implementar verificaciones antes de cada acceso, y diseñar la arquitectura del programa para minimizar el tiempo entre la liberación y la invalidación de referencias.

Las herramientas de análisis estático y dinámico pueden detectar muchos casos de este tipo de errores durante el desarrollo y las pruebas, proporcionando una capa adicional de protección contra estos defectos.

Riesgos asociados

- Corrupción de memoria debido a la escritura o lectura en direcciones ya desasignadas.
- Condiciones de carrera y errores de sincronización en aplicaciones concurrentes.
- Generación de resultados inesperados o incoherentes en la ejecución del programa.
- Fallas de segmentación y caída inesperada del programa (segmentation fault o core dump).
- Vulnerabilidad a ataques de seguridad, como ejecución de código arbitrario mediante el uso de punteros colgantes (use-after-free).
- Exposición de información sensible almacenada previamente en esa memoria.
- Dificultad para la depuración y el mantenimiento debido a errores intermitentes y no determinísticos.
- Inestabilidad general en el sistema o servicio afectado.

Buenas prácticas

- Asignar `NULL` inmediatamente después de liberar memoria para evitar referencias colgantes.

```
//-- Uso de puntero después de liberar memoria
char *ptr = malloc(20);
free(ptr);
ptr[0] = 'A';
```

```
//-- Liberación y posterior invalidación del puntero
char *ptr2 = malloc(20);
free(ptr2);
ptr2 = NULL;
```

- Comprobar que el puntero no sea `NULL` antes de acceder a la memoria.

```
//-- Acceso a memoria potencialmente liberada
char *buffer = malloc(10);
free(buffer);
if (buffer[0] == 'A') { }
```

```
//-- Validación del puntero antes de uso
char *buffer2 = malloc(10);
free(buffer2);
buffer2 = NULL;
if (buffer2 != NULL && buffer2[0] == 'A') { }
```

- Evitar múltiples liberaciones del mismo puntero mediante control explícito del estado.

```
//-- Doble liberación del mismo bloque de memoria
char *data = malloc(30);
free(data);
free(data);
```

```
//-- Control de liberación única del recurso
char *data2 = malloc(30);
if (data2 != NULL) {
  free(data2);
  data2 = NULL;
}
```

- Limitar el ámbito del puntero para reducir el riesgo de uso tras liberación.

```
//-- Puntero con ámbito amplio reutilizado tras liberación
char *msg;
msg = malloc(15);
free(msg);
printf("%s", msg);
```

```
//-- Puntero limitado a un bloque específico
char *msg2 = malloc(15);
free(msg2);
msg2 = NULL;
```

2.9.1.2 Estándares metodológicos

- **CERT/C-MEM30**: Do not access freed memory.

2.9.2 [c.mem.2] - No acceder a memoria no inicializada

ID	c.mem.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El acceso a memoria no inicializada constituye una fuente frecuente de vulnerabilidades de seguridad. Las variables automáticas y la memoria asignada dinámicamente contienen valores impredecibles que pueden provocar fallos de ejecución, filtración de información sensible o permitir ataques de explotación. Esta práctica compromete el **determinismo del programa** y genera resultados inconsistentes entre diferentes ejecuciones y plataformas.

Codificación insegura

Uso de una variable local sin inicializar para calcular y mostrar un valor. La intención es observar cómo el programa compila y ejecuta, pero utiliza memoria que no ha sido inicializada previamente, lo que puede producir resultados impredecibles y comportamiento indefinido.

```
#include <stdio.h>

//-- Programa que usa una variable sin inicializar
int main() {
  int numero;
  int resultado;

  //-- Se utiliza la variable 'numero' sin haber sido inicializada
  resultado = numero + 10;

  //-- Se muestra el resultado en pantalla
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

Codificación recomendada

Inicialización explícita de las variables antes de su uso en operaciones aritméticas. La intención es garantizar que toda la memoria utilizada tenga un valor definido, evitando comportamiento indefinido y mejorando la previsibilidad y seguridad del programa.

```
#include <stdio.h>

//-- Programa que inicializa correctamente las variables
int main() {
  int numero = 0;
  int resultado = 0;

  //-- Se asigna un valor válido antes de utilizar la variable
  numero = 5;

  //-- Se realiza la operación con variables inicializadas
  resultado = numero + 10;

  //-- Se muestra el resultado en pantalla
  printf("Resultado: %d\n", resultado);

  return 0;
}
```

2.9.2.1 Descripción

El acceso a memoria no inicializada constituye una vulnerabilidad común que ocurre cuando el programa lee valores de variables o estructuras de datos antes de asignarles un valor definido. Esta situación genera comportamiento indefinido, ya que el contenido de la memoria puede contener datos residuales de operaciones previas o valores aleatorios.

La memoria no inicializada presenta **contenido impredecible** que varía según el estado previo del sistema, la implementación del compilador y las condiciones de ejecución. Este comportamiento no determinista compromete la fiabilidad del programa y puede exponer información sensible que permanecía en memoria de sesiones anteriores.

Los riesgos asociados incluyen **comportamiento errático del programa**, decisiones lógicas basadas en datos incorrectos, y potencial filtración de información confidencial. En contextos de seguridad, el acceso a memoria no inicializada puede revelar datos de otros procesos o sesiones previas del mismo programa.

Las variables automáticas, los arrays dinámicos y las estructuras asignadas mediante `malloc` son particularmente susceptibles a este problema, ya que no reciben inicialización automática. Las variables globales y estáticas, por el contrario, se inicializan implícitamente a cero.

La mitigación efectiva requiere **inicialización explícita** de todas las variables antes de su primer uso. Para memoria dinámica, se recomienda el uso de funciones como `calloc` que proporcionan inicialización automática, o la aplicación manual de `memset` tras la asignación con `malloc`.

Los compiladores modernos ofrecen opciones de diagnóstico que detectan algunos casos de uso de variables no inicializadas, aunque estas herramientas presentan limitaciones en escenarios complejos que involucran punteros o acceso indirecto a memoria.

Riesgos asociados

- Lectura de valores indeterminados que pueden causar un comportamiento inesperado del programa.
- Ejecución de código con resultados erróneos o no deterministas.
- Divulgación de información sensible almacenada previamente en la memoria.
- Exposición a vulnerabilidades de seguridad como fuga de información (information disclosure).
- Posibilidad de crash o fallo del programa (por ejemplo, segmentation fault).
- Dificultad para depurar y reproducir errores debido a la aleatoriedad de los valores en memoria no inicializada.
- Potencial aprovechamiento de la condición por un atacante para ejecutar ataques como control del flujo del programa o escalada de privilegios.
- Incremento del riesgo de vulnerabilidades de corrupción de memoria (memory corruption).

Buenas prácticas

- Inicializar explícitamente todas las variables locales antes de su uso.

```
//-- Variable local sin inicializar utilizada en una condición
int flag;
if (flag == 1) {
  procesar();
}
```

```
//-- Variable inicializada antes de cualquier uso
int flag = 0;
if (flag == 1) {
  procesar();
}
```

- Inicializar la memoria dinámica inmediatamente después de reservarla.

```
//-- Memoria reservada sin inicializar utilizada directamente
int *buffer = malloc(10 * sizeof(int));
int suma = buffer[0];
```

```
//-- Memoria reservada e inicializada antes de su uso
int *buffer = malloc(10 * sizeof(int));
if (buffer != NULL) {
  memset(buffer, 0, 10 * sizeof(int));
  int suma = buffer[0];
}
```

- Inicializar completamente estructuras y arrays antes de acceder a sus campos.

```
//-- Estructura declarada sin inicializar accediendo a sus campos
struct Datos { int id; char tipo; };
struct Datos d;
if (d.id > 0) {
  procesar(d.tipo);
}
```

```
//-- Estructura inicializada al momento de su declaración
struct Datos { int id; char tipo; };
struct Datos d = {0};
if (d.id > 0) {
  procesar(d.tipo);
}
```

2.9.2.2 Estándares metodológicos

- **CERT/C-EXP33**: Do not read uninitialized memory.

2.9.3 [c.mem.3] - Liberar la memoria asignada dinámicamente cuando deje de ser necesaria

ID	c.mem.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La gestión inadecuada de la memoria dinámica genera **memory leaks** que degradan el rendimiento y pueden agotar los recursos del sistema. Cuando asignamos memoria mediante `malloc()`, `calloc()` o `realloc()` y no la liberamos con `free()`, esta permanece reservada hasta la finalización del programa, acumulándose progresivamente y causando problemas de estabilidad en aplicaciones de larga duración.

Codificación insegura

Uso de memoria dinámica para almacenar una cadena introducida por el usuario y mostrarla por pantalla. Se observa la gestión del ciclo de vida del recurso reservado y qué ocurre cuando la memoria deja de utilizarse dentro del flujo del programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//-- Punto de entrada del programa
int main(void) {
  //-- Reserva dinámica de memoria para 100 caracteres
  char *buffer = (char *)malloc(100 * sizeof(char));

  //-- Verificación básica de error en la reserva
  if (buffer == NULL) {
    printf("Error al reservar memoria\n");
    return 1;
  }
  //-- Solicita una cadena al usuario
  printf("Introduce una cadena: ");
  fgets(buffer, 100, stdin);

  //-- Muestra la cadena introducida
  printf("Cadena introducida: %s\n", buffer);

  //-- El programa finaliza sin liberar la memoria reservada
  return 0;
}
```

Codificación recomendada

Uso de memoria dinámica para almacenar una cadena introducida por el usuario y mostrarla por pantalla, gestionando correctamente la liberación del recurso cuando deja de ser necesario, evitando fugas de memoria y manteniendo un ciclo de vida controlado.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//-- Punto de entrada del programa
int main(void) {
  //-- Reserva dinámica de memoria para 100 caracteres
  char *buffer = (char *)malloc(100 * sizeof(char));

  //-- Verificación básica de error en la reserva
  if (buffer == NULL) {
    printf("Error al reservar memoria\n");
```

```
    return 1;
  }
  //-- Solicita una cadena al usuario
  printf("Introduce una cadena: ");
  fgets(buffer, 100, stdin);

  //-- Muestra la cadena introducida
  printf("Cadena introducida: %s\n", buffer);

  //-- Liberación explícita de la memoria cuando deja de ser necesaria
  free(buffer);

  //-- Asignación a NULL para evitar punteros colgantes
  buffer = NULL;

  //-- Finalización correcta del programa
  return 0;
}
```

2.9.3.1 Descripción

La gestión manual de memoria dinámica requiere que toda memoria asignada mediante funciones como `malloc()`, `calloc()` o `realloc()` sea explícitamente liberada con `free()` cuando ya no se necesite. Esta responsabilidad recae completamente en el programador, ya que no existe un mecanismo automático de recolección de basura.

El incumplimiento de esta regla genera **memory leaks** o fugas de memoria, donde los bloques asignados permanecen reservados durante toda la ejecución del programa sin posibilidad de reutilización. En aplicaciones de larga duración o que realizan asignaciones frecuentes, estas fugas pueden consumir progresivamente toda la memoria disponible del sistema.

La detección de estas fugas presenta desafíos particulares porque sus efectos no son inmediatamente visibles. Los síntomas suelen manifestarse como degradación gradual del rendimiento, fallos en asignaciones posteriores o, en casos extremos, terminación del programa por agotamiento de memoria.

La mitigación efectiva requiere establecer una correspondencia estricta entre cada llamada a funciones de asignación y su respectiva liberación. Esto implica identificar claramente el **ciclo de vida** de cada bloque de memoria y determinar el punto exacto donde debe ser liberado. Las herramientas de análisis estático y dinámico, como Valgrind o los sanitizers del compilador, facilitan la detección de estas fugas durante el desarrollo y las pruebas.

La implementación de patrones de gestión de recursos, como la asignación y liberación en el mismo nivel de abstracción, reduce significativamente la probabilidad de omitir liberaciones necesarias. La documentación clara de la responsabilidad de liberación en

interfaces y funciones también contribuye a prevenir estos errores en equipos de desarrollo.

Riesgos asociados

- Pérdida de memoria (memory leak): la aplicación consume cada vez más memoria, deteriorando el rendimiento.
- Agotamiento de memoria: el sistema puede quedarse sin memoria, provocando fallos en la aplicación u otros procesos.
- Vulnerabilidades de denegación de servicio (DoS): un atacante puede explotar la falta de liberación de memoria para provocar que la aplicación o el sistema se bloquee.
- Incremento en el tiempo de respuesta: tanto la fragmentación como la acumulación de memoria no liberada ralentizan la aplicación.
- Pérdida de escalabilidad: la aplicación no puede manejar una cantidad elevada de transacciones o usuarios debido al mal uso de recursos.
- Imposibilidad de recuperación: algunos recursos asignados nunca se liberan, haciendo imposible que la aplicación se recupere sin reiniciar.
- Posibles pérdidas de datos: si la aplicación falla por falta de memoria, puede perder datos almacenados en la sesión actual.
- Fragmentación de memoria: la memoria disponible se fragmenta, lo que complica nuevas asignaciones de bloques grandes.
- Compromiso de seguridad: los errores de gestión de memoria pueden facilitar que un atacante explote otras vulnerabilidades relacionadas, como buffer overflows.

Buenas prácticas

- Liberar siempre la memoria dinámica cuando deje de ser necesaria para evitar fugas de memoria.

```
//-- Reserva memoria sin liberarla, provocando fuga
char *buffer = malloc(128);
if (buffer == NULL) {
  return;
}
strcpy(buffer, "dato");
```

```
//-- Libera la memoria después de usarla
char *buffer2 = malloc(128);
if (buffer2 == NULL) {
  return;
```

```
}
strcpy(buffer2, "dato");
free(buffer2);
```

- Asignar `NULL` al puntero después de liberar la memoria para evitar referencias colgantes.

```
//-- Libera memoria, pero deja el puntero apuntando a memoria liberada
char *data = malloc(64);
if (data == NULL) {
  return;
}
free(data);
strcpy(data, "uso inseguro");
```

```
//-- Libera memoria y anula el puntero para evitar uso posterior
char *data2 = malloc(64);
if (data2 == NULL) {
  return;
}
free(data2);
data2 = NULL;
```

- Liberar todos los recursos previamente asignados en caso de error antes de salir de la función.

```
//-- Sale por error sin liberar recursos previamente reservados
char *a = malloc(32);
char *b = malloc(32);
if (a == NULL || b == NULL) {
  return;
}
```

```
//-- Libera recursos ya asignados antes de salir por error
char *x = malloc(32);
if (x == NULL) {
  return;
}
char *y = malloc(32);
if (y == NULL) {
  free(x);
  return;
}
free(y);
free(x);
```

- Evitar la doble liberación de memoria controlando el flujo del programa.

```
//-- Libera dos veces el mismo puntero
char *ptr = malloc(16);
if (ptr == NULL) {
  return;
}
free(ptr);
free(ptr);
```

```
//-- Controla la liberación para que ocurra una sola vez
char *ptr2 = malloc(16);
if (ptr2 == NULL) {
  return;
}
free(ptr2);
ptr2 = NULL;
```

2.9.3.2 Estándares metodológicos

- **CERT/C-MEM31**: Free dynamically allocated memory when no longer needed.

2.9.4 [c.mem.4] - No liberar memoria no asignada dinámicamente

ID	c.mem.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La liberación incorrecta de memoria constituye una fuente frecuente de errores críticos en programas. Intentar liberar memoria que no fue asignada dinámicamente mediante `malloc()`, `calloc()` o `realloc()` puede causar **corrupción del heap, fallos de segmentación o vulnerabilidades de seguridad** explotables.

Codificación insegura

Se muestra un programa que gestiona memoria para almacenar un número entero. La intención es ilustrar cómo puede producirse un comportamiento indefinido cuando se intenta liberar memoria que no fue reservada dinámicamente, lo que puede provocar fallos en tiempo de ejecución o vulnerabilidades.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  //-- Se declara una variable en la pila (memoria automática)
  int valor = 10;
```

```
  //-- Se obtiene un puntero a una variable que NO fue reservada con malloc
  int *ptr = &valor;

  //-- Se muestra el valor apuntado
  printf("Valor: %d\n", *ptr);

  //-- Error: se intenta liberar memoria que no fue asignada dinámicamente
  free(ptr);

  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Se presenta un programa que reserva memoria dinámicamente para almacenar un número entero y, posteriormente, la libera correctamente. El objetivo es mostrar el uso adecuado de `malloc` y `free`, asegurando que solo se libere memoria previamente asignada dinámicamente y evitando comportamientos indefinidos.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  //-- Reserva dinámica de memoria para un entero
  int *ptr = (int *)malloc(sizeof(int));

  //-- Verificación de que la reserva fue exitosa
  if (ptr == NULL) {
    //-- Manejo de error si malloc falla
    fprintf(stderr, "Error al reservar memoria\n");
    return 1;
  }
  //-- Asignación de valor a la memoria reservada
  *ptr = 20;
  printf("Valor: %d\n", *ptr);

  //-- Liberación correcta de la memoria asignada dinámicamente
  free(ptr);
  ptr = NULL;

  //-- Fin del programa
  return 0;
}
```

2.9.4.1 Descripción

Esta regla prohíbe llamar a las funciones de liberación de memoria como `free()`, `realloc()` o `aligned_alloc()` sobre punteros que no apuntan a memoria asignada dinámicamente mediante `malloc()`, `calloc()`, `realloc()` o funciones equivalentes.

El comportamiento está **indefinido** cuando se intenta liberar memoria que no fue asignada dinámicamente, incluyendo variables automáticas, estáticas, miembros de estructuras, elementos de arrays o cualquier dirección de memoria obtenida por otros medios. Esta violación puede provocar corrupción del heap, fallos de segmentación o comportamientos impredecibles del programa.

Los errores más comunes incluyen intentar liberar variables locales declaradas en la pila, liberar la misma dirección de memoria múltiples veces, o liberar punteros que han sido modificados aritméticamente desde su asignación original. También ocurre cuando se confunde el origen de un puntero o cuando se libera memoria que forma parte de una estructura de datos más grande.

La mitigación efectiva requiere mantener un **control estricto del ciclo de vida** de los punteros y establecer correspondencias claras entre cada llamada de asignación y su respectiva liberación. Es necesario verificar el origen de cada puntero antes de liberarlo y asegurar que solo se liberen direcciones obtenidas directamente de funciones de asignación dinámica.

Las herramientas de análisis estático y dinámico pueden detectar muchas violaciones de esta regla, especialmente cuando se combinan con técnicas de seguimiento de memoria y validación de punteros durante el desarrollo y las pruebas.

Riesgos asociados

- Corrupción de memoria: la liberación de memoria no asignada dinámicamente puede sobrescribir estructuras internas del heap, causando corrupción en la gestión de memoria del programa.
- Errores de acceso: puede provocar errores de segmentación (segmentation fault), resultando en la terminación inesperada del programa.
- Comportamiento indefinido: liberar memoria no asignada produce un comportamiento indefinido, lo que dificulta el diagnóstico y la depuración de errores.
- Vulnerabilidades de seguridad: un atacante podría explotar el comportamiento indefinido para ejecutar código malicioso o provocar denegación de servicio (DoS).
- Fugas de memoria: un mal manejo de la memoria puede camuflar fugas de memoria, incrementando el consumo innecesario de recursos.

Buenas prácticas

- Inicializar los punteros y liberar únicamente memoria que haya sido asignada dinámicamente.

```
//-- Liberación de puntero no inicializado
int *ptr;
free(ptr);
```

```
//-- Liberación después de asignación dinámica válida
int *ptr2 = NULL;
ptr2 = malloc(sizeof(int));
if (ptr2 != NULL) {
  free(ptr2);
  ptr2 = NULL;
}
```

- No llamar a `free()` sobre variables automáticas (stack).

```
//-- Liberación de variable automática
int value = 10;
free(&value);
```

```
//-- Uso de memoria dinámica para permitir liberación segura
int *valuePtr = malloc(sizeof(int));
if (valuePtr != NULL) {
  *valuePtr = 10;
  free(valuePtr);
  valuePtr = NULL;
}
```

- No liberar memoria que no apunta al inicio del bloque asignado.

```
//-- Liberación de puntero desplazado
char *buffer = malloc(10);
if (buffer != NULL) {
  char *offset = buffer + 5;
  free(offset);
}
```

```
//-- Liberación usando el puntero original devuelto por malloc
char *buffer2 = malloc(10);
if (buffer2 != NULL) {
  char *offset2 = buffer2 + 5;
  free(buffer2);
  buffer2 = NULL;
```

```
}
```

- No liberar memoria gestionada por bibliotecas o no obtenida mediante funciones de asignación dinámica.

```
//-- Liberación de memoria no asignada dinámicamente (cadena literal)
char *text = "Hola";
free(text);
```

```
//-- Uso de copia dinámica antes de liberar
char *text2 = malloc(strlen("Hola") + 1);
if (text2 != NULL) {
  strcpy(text2, "Hola");
  free(text2);
  text2 = NULL;
}
```

2.9.4.2 Estándares metodológicos

- **CERT/C-MEM34**: Only free memory allocated dynamically.

2.9.5 [c.mem.5] - Garantizar que se asigne memoria suficiente para un objeto

ID	c.mem.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La asignación insuficiente de memoria para objetos puede provocar **desbordamientos de buffer** y corrupción de datos adyacentes en el heap. Este problema surge cuando se calcula incorrectamente el tamaño necesario para almacenar estructuras, arrays o cadenas, especialmente al usar funciones como `malloc()`, `calloc()` o `realloc()`. Los errores de cálculo pueden derivar de multiplicaciones que causan desbordamiento aritmético o de no considerar el espacio adicional requerido para terminadores nulos en cadenas.

Codificación insegura

Se muestra la asignación dinámica de memoria para almacenar una cadena introducida por el usuario. La intención es evidenciar qué ocurre cuando no se reserva suficiente espacio para el objeto, especialmente cuando se omite espacio para el carácter nulo de terminación, y cómo esto puede provocar desbordamientos y comportamiento indefinido.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

int main() {
  //-- Cadena de ejemplo que simula entrada del usuario
  const char *input = "Seguridad";

  //-- Se reserva memoria sin considerar el caracter nulo final
  char *buffer = (char *)malloc(strlen(input));
  if (buffer == NULL) {
    //-- Validación básica de error de memoria
    return 1;
  }
  //-- Copia de la cadena incluyendo el caracter nulo, provocando desbordamiento
  strcpy(buffer, input);

  //-- Uso del buffer potencialmente corrupto
  printf("Cadena copiada: %s\n", buffer);

  //-- Liberación de memoria
  free(buffer);

  return 0;
}
```

Codificación recomendada

Se muestra la asignación dinámica de memoria asegurando que el tamaño reservado cubre completamente el objeto, incluyendo el carácter nulo de terminación. La intención es demostrar cómo calcular correctamente el tamaño requerido y validar la asignación para evitar desbordamientos y comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  //-- Cadena de ejemplo que simula entrada del usuario
  const char *input = "Seguridad";

  //-- Se calcula el tamaño incluyendo el caracter nulo final
  size_t length = strlen(input) + 1;

  //-- Reserva de memoria suficiente para la cadena completa
  char *buffer = (char *)malloc(length);
  if (buffer == NULL) {
    //-- Validación de error en la asignación de memoria
    return 1;
```

```
    }
    //-- Copia segura de la cadena dentro del límite reservado
    strcpy(buffer, input);

    //-- Uso seguro del buffer correctamente dimensionado
    printf("Cadena copiada: %s\n", buffer);

    //-- Liberación de memoria
    free(buffer);

    return 0;
}
```

2.9.5.1 Descripción

La asignación de memoria insuficiente para un objeto constituye una vulnerabilidad que puede comprometer la integridad y estabilidad del programa. Esta situación ocurre cuando el tamaño especificado en las funciones de asignación dinámica como `malloc`, `calloc` o `realloc` es menor al espacio realmente necesario para almacenar el objeto o estructura de datos.

El problema surge frecuentemente al calcular el tamaño requerido para arrays, estructuras complejas o buffers de datos. Los errores de cálculo pueden derivar de operaciones aritméticas incorrectas, uso de tipos de datos inadecuados para representar tamaños, o falta de consideración de elementos adicionales como terminadores nulos en cadenas de caracteres.

La asignación insuficiente genera **desbordamientos de buffer** cuando el programa intenta escribir más datos de los que el espacio asignado puede contener. Estos desbordamientos pueden corromper estructuras de datos adyacentes en memoria, provocar comportamientos impredecibles del programa, o crear vulnerabilidades explotables por atacantes.

Los riesgos se intensifican en operaciones que involucran arrays multidimensionales, estructuras con miembros de tamaño variable, o cuando se realizan cálculos de tamaño basados en entrada de usuario sin validación previa. La corrupción de memoria resultante puede manifestarse como fallos de segmentación, comportamiento errático del programa, o vulnerabilidades de seguridad.

La mitigación efectiva requiere **verificación rigurosa de los cálculos de tamaño** antes de realizar asignaciones de memoria. Esto incluye validar que las operaciones aritméticas no produzcan desbordamientos, usar tipos de datos apropiados para representar tamaños, y considerar todos los elementos que componen el objeto a almacenar.

Las herramientas de análisis estático pueden detectar patrones comunes de asignación insuficiente, mientras que los analizadores dinámicos de memoria identifican desbordamientos durante la ejecución. La implementación de verificaciones de límites

y el uso de funciones de asignación que inicialicen la memoria contribuyen a reducir los riesgos asociados.

Riesgos asociados

- Desbordamiento de buffer (buffer overflow).
- Escritura fuera de los límites del objeto.
- Lectura de memoria no asignada.
- Corrupción de memoria.
- Comportamiento indefinido del programa.
- Exposición o fuga de información sensible.
- Ejecución de código arbitrario.
- Condiciones de denegación de servicio (DoS).
- Errores de segmentación (segmentation fault).
- Vulnerabilidades explotables por atacantes externos.

Buenas prácticas

- Calcular el tamaño de memoria usando `sizeof(tipo)` en lugar de valores constantes para evitar asignaciones insuficientes.

```
//-- Reserva memoria usando un tamaño fijo que puede ser insuficiente
char *buffer = (char *)malloc(10);
strcpy(buffer, "Texto demasiado largo");
```

```
//-- Reserva memoria basada en el tamaño real del tipo
char *buffer = (char *)malloc(sizeof(char) * 20);
strcpy(buffer, "Texto seguro");
```

- Asignar memoria usando `sizeof(*puntero)` para evitar errores cuando cambia el tipo de dato.

```
//-- Usa un tamaño explícito que puede no coincidir con el tipo real
int *ptr = (int *)malloc(10 * sizeof(long));
```

```
//-- Usa el tamaño referenciado por el puntero para mayor consistencia
int *ptr = (int *)malloc(10 * sizeof(*ptr));
```

- Considerar el espacio adicional necesario para terminadores u otros campos implícitos.

```
//-- Omite espacio para el caracter nulo final
char *cadena = (char *)malloc(strlen("ejemplo") * sizeof(char));
strcpy(cadena, "ejemplo");
```

```
//-- Incluye espacio adicional para el terminador nulo
char *cadena = (char *)malloc((strlen("ejemplo") + 1) * sizeof(char));
strcpy(cadena, "ejemplo");
```

2.9.5.2 Estándares metodológicos

- **CERT/C-MEM35**: Allocate sufficient memory for an object.

2.10 Validaciones

2.10.1 [c.val.1] - No incluir entrada del usuario en cadenas de formato

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

Las funciones de formato como `printf` y `scanf` interpretan secuencias especiales en sus cadenas de formato que pueden ejecutar operaciones no previstas. Cuando se permite que una **entrada del usuario** forme parte directa de estas cadenas, los atacantes pueden insertar especificadores de formato maliciosos para leer o escribir memoria arbitraria, causando **vulnerabilidades de seguridad críticas**.

Codificación insegura

Se muestra un programa que lee una cadena desde la entrada estándar y la imprime por pantalla. El objetivo es observar cómo el uso directo de datos externos como cadena de formato en funciones como printf puede generar vulnerabilidades de tipo format string y comportamientos inesperados.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char buffer[100];

  //-- Solicita una cadena al usuario
  printf("Introduce un texto: ");
  if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
    //-- Error al leer la entrada
    return 1;
  }
```

```
  //-- Elimina el salto de línea si existe
  buffer[strcspn(buffer, "\n")] = '\0';

  //-- Uso inseguro: la entrada del usuario se usa como cadena de formato
  printf(buffer);
  printf("\n");

  return 0;
}
```

Codificación recomendada

Se muestra un programa que lee una cadena desde la entrada estándar y la imprime por pantalla utilizando una cadena de formato constante. El objetivo es aprender a separar los datos de la cadena de formato, evitando que la entrada externa controle el comportamiento de funciones de impresión.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char buffer[100];

  //-- Solicita una cadena al usuario
  printf("Introduce un texto: ");
  if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
    //-- Error al leer la entrada
    return 1;
  }
  //-- Elimina el salto de línea si existe
  buffer[strcspn(buffer, "\n")] = '\0';

  //-- Uso seguro: se define explícitamente la cadena de formato
  printf("%s", buffer);
  printf("\n");

  return 0;
}
```

2.10.1.1 Descripción

Las funciones de formato como `printf`, `sprintf`, `fprintf` y sus variantes interpretan secuencias especiales en las cadenas de formato para determinar cómo procesar y mostrar los argumentos. Cuando se permite que datos controlados por el usuario formen parte de estas cadenas de formato, se crea una vulnerabilidad que puede ser explotada para ejecutar código arbitrario o acceder a información sensible de la memoria.

El problema surge porque las funciones de formato procesan especificadores como `%s`, `%d`, `%x` y `%n` de manera dinámica. Si un atacante puede inyectar estos especificadores a través de entrada no validada, puede manipular el comportamiento de la función para leer o escribir en ubicaciones de memoria no autorizadas.

Los **ataques de cadena de formato** representan una clase seria de vulnerabilidades que pueden resultar en divulgación de información, corrupción de memoria o ejecución de código malicioso. Estos ataques aprovechan el mecanismo interno de las funciones de formato que acceden a la pila para obtener los argumentos correspondientes a cada especificador.

La mitigación efectiva requiere tratar toda entrada del usuario como datos, nunca como instrucciones de formato. Las cadenas de formato deben ser constantes controladas por el desarrollador, mientras que los datos del usuario deben pasarse exclusivamente como argumentos a estas funciones. Las herramientas de análisis estático pueden detectar muchos casos donde se viola esta regla y los compiladores modernos incluyen advertencias específicas para estos patrones peligrosos.

Esta vulnerabilidad es particularmente insidiosa porque el código puede funcionar correctamente en condiciones normales, pero fallar catastróficamente cuando se enfrenta a entrada maliciosa. La validación de entrada por sí sola no es suficiente, ya que puede ser difícil anticipar todas las combinaciones posibles de especificadores de formato que un atacante podría intentar inyectar.

Riesgos asociados

- Ejecución de ataques de formato de cadena ("%n", "%s", etc.) que pueden permitir la modificación de memoria.
- Riesgo de corrupción de memoria/aplicación (buffer overflow, memory leak).
- Divulgación inadvertida de información sensible a través de la impresión de direcciones o contenido de memoria (information disclosure).
- Posibilidad de ejecución remota de código si el atacante controla la cadena de formato.
- Interrupciones inesperadas del programa (crash, denegación de servicio).
- Dificultad de detectar y depurar errores en tiempo de ejecución causados por entradas maliciosas.
- Aumento de vectores para explotación de otras vulnerabilidades debido a corrupción de memoria.

Buenas prácticas

- Validar y sanitizar cualquier dato externo antes de usarlo en funciones de formato.

```
//-- Uso directo de entrada externa como cadena de formato
char buffer[128];
snprintf(buffer, sizeof(buffer), user_input);
```

```
//-- Uso de formato fijo y datos externos como argumento
char buffer[128];
snprintf(buffer, sizeof(buffer), "%s", user_input);
```

- Definir explícitamente especificadores de formato en funciones printf-like.

```
//-- Construcción dinámica insegura de formato
printf(user_input);
```

```
//-- Especificación explícita del formato esperado
printf("%s", user_input);
```

- Restringir el uso de especificadores peligrosos como %n en cadenas de formato.

```
//-- Cadena controlada externamente que podría incluir %n
fprintf(log_file, user_input);
```

```
//-- Formato controlado que evita interpretación de especificadores arbitrarios
fprintf(log_file, "%s", user_input);
```

- Utilizar funciones seguras con límite de tamaño y formato constante.

```
//-- Uso de sprintf con formato no controlado
char buffer[64];
sprintf(buffer, user_input);
```

```
//-- Uso de snprintf con formato fijo y límite de tamaño
char buffer[64];
snprintf(buffer, sizeof(buffer), "%s", user_input);
```

- Separar la lógica de construcción de mensajes del contenido proporcionado por el usuario.

```
//-- Concatenación directa en función de impresión
printf(user_input);
```

```
//-- Construcción previa del mensaje con formato constante
char mensaje[128];
snprintf(mensaje, sizeof(mensaje), "Mensaje: %s", user_input);
```

```
printf("%s", mensaje);
```

2.10.1.2 Estándares metodológicos

- **CERT/C-FIO30**: Exclude user input from format strings.

2.10.2 [c.val.2] - Distinguir correctamente entre caracteres leídos y EOF o WEOF

ID	c.val.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de entrada de caracteres pueden devolver valores que representan tanto caracteres válidos como condiciones de error o fin de archivo. La confusión entre estos valores puede provocar **interpretaciones incorrectas** de los datos leídos, especialmente cuando el valor de `EOF` o `WEOF` coincide con un carácter válido tras conversiones de tipo inadecuadas.

Codificación insegura

Lectura de caracteres desde la entrada estándar utilizando `getchar()` para procesar datos hasta encontrar el fin de fichero. El objetivo es comprender cómo el tipo de dato usado para almacenar el valor retornado influye en la correcta detección de `EOF` y evitar bucles infinitos o errores lógicos cuando se procesan flujos de entrada.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Se utiliza char para almacenar el valor de getchar()
  char c;

  //-- Mensaje informativo para el usuario
  printf("Introduce texto y finaliza con EOF (Ctrl+D o Ctrl+Z):\n");

  //-- Bucle que intenta leer hasta EOF
  while ((c = getchar()) != EOF) {
    //-- Se imprime el carácter leído
    putchar(c);
  }
  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Lectura segura de caracteres desde la entrada estándar almacenando el valor retornado por `getchar()` en un tipo adecuado, permitiendo diferenciar cualquier valor válido de carácter del valor especial `EOF`. Se busca garantizar una detección correcta del fin de fichero y evitar comportamientos indefinidos o bucles infinitos.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Se utiliza int para almacenar correctamente cualquier carácter o EOF
  int c;

  //-- Mensaje informativo para el usuario
  printf("Introduce texto y finaliza con EOF (Ctrl+D o Ctrl+Z):\n");

  //-- Bucle que lee hasta detectar explícitamente EOF
  while ((c = getchar()) != EOF) {
    //-- Conversión explícita a char solo después de validar que no es EOF
    putchar((char)c);
  }
  //-- Fin del programa
  return 0;
}
```

2.10.2.1 Descripción

La función `getchar()` y funciones similares de entrada de caracteres devuelven un valor de tipo `int` en lugar de `char` por una razón específica: necesitan poder representar tanto todos los caracteres válidos como el valor especial **EOF** (End of File). Esta distinción resulta crítica para el manejo correcto de flujos de entrada y la detección de condiciones de error o fin de archivo.

El problema surge cuando se almacena el valor de retorno en una variable de tipo `char` en lugar de `int`. Dado que `char` tiene un rango limitado, puede producirse una **truncado de datos** que impida distinguir entre un carácter válido y la condición EOF. Esta confusión puede llevar a bucles infinitos, procesamiento incorrecto de datos o fallos en la detección del final del flujo de entrada.

El valor EOF se define típicamente como -1, pero su representación exacta depende de la implementación. Cuando este valor se trunca al almacenarlo en un `char`, puede coincidir con el valor de un carácter válido, especialmente en sistemas donde `char` es `unsigned` por defecto. Esta ambigüedad elimina la capacidad del programa para determinar si se ha leído un carácter legítimo o si se ha alcanzado el final del archivo.

Las funciones de entrada de caracteres anchos presentan el mismo problema con **WEOF**, que debe distinguirse de los caracteres anchos válidos. El principio de almacenar el valor

de retorno en el tipo apropiado (`wint_t` para caracteres anchos) aplica de manera similar.

La mitigación de este problema requiere el uso consistente de variables del tipo correcto para almacenar los valores de retorno de las funciones de entrada. Esto preserva la información completa necesaria para realizar comparaciones precisas con EOF o WEOF antes de procesar el carácter como dato válido.

Riesgos asociados

- Confusión entre el valor de un carácter válido y el valor de EOF/WEOF, lo que puede causar salidas incorrectas o comportamientos inesperados.
- Lectura errónea del flujo de entrada, omitiendo o procesando incorrectamente datos válidos.
- Condiciones de carrera y bucles infinitos al interpretar erróneamente el fin de un archivo o flujo de datos.
- Desbordamientos de buffer al almacenar EOF/WEOF (normalmente -1) como si fuera un carácter válido (por ejemplo, en variables de tipo `char`).
- Errores de lógica que pueden provocar la pérdida de información durante la manipulación de archivos o flujos.
- Vulnerabilidad ante ataques de denegación de servicio (DoS) si se producen bucles o comportamientos extremos inesperados.
- Posibilidad de fallos por acceso fuera de los límites del buffer, debido a interpretaciones incorrectas del valor leído.
- Compromiso en la interpretación y validación de datos recibidos, facilitando que un atacante induzca errores en la ejecución del programa.

Buenas prácticas

- Utilizar un tipo entero (`int` o `wint_t`) para almacenar el valor devuelto por funciones como `getchar()` o `fgetwc()`, evitando almacenarlo en tipos `char` o `wchar_t`.

```
//-- Almacenar el resultado de getchar() en char puede confundir EOF con un
carácter válido
char c;
c = getchar();
if (c == EOF) {
  /* manejo */
}
```

```
//-- Usar int para preservar correctamente el valor especial EOF
int c2;
```

```
c2 = getchar();
if (c2 == EOF) {
  /* manejo */
}
```

- Comparar explícitamente contra `EOF` o `WEOF` antes de convertir o procesar el carácter leído.

```
//-- Convertir a char antes de verificar EOF puede perder el valor especial
int r;
char ch;
r = fgetc(stdin);
ch = (char)r;
if (ch != EOF) {
  procesar(ch);
}
```

```
//-- Verificar primero contra EOF y luego convertir para procesar
int r2;
r2 = fgetc(stdin);
if (r2 != EOF) {
  char ch2 = (char)r2;
  procesar(ch2);
}
```

- En lectura de caracteres anchos, usar `wint_t` y comparar con `WEOF` en lugar de `wchar_t`.

```
//-- Usar wchar_t para almacenar fgetwc() puede impedir distinguir WEOF
wchar_t wc;
wc = fgetwc(stdin);
if (wc == WEOF) {
  /* manejo */
}
```

```
//-- Usar wint_t permite identificar correctamente WEOF
wint_t wc2;
wc2 = fgetwc(stdin);
if (wc2 == WEOF) {
  /* manejo */
}
```

2.10.2.2 Estándares metodológicos

- **CERT/C-FIO34**: Distinguish between characters read from a file and EOF or WEOF.

2.10.3 [c.val.3] - Evitar condiciones de carrera TOCTOU al acceder a archivos

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

Las operaciones de acceso a archivos que separan la verificación de permisos o existencia del uso posterior del archivo crean ventanas temporales donde otros procesos pueden modificar el sistema de archivos. Estas **condiciones de carrera TOCTOU** (Time-of-Check to Time-of-Use) permiten que atacantes manipulen archivos entre la comprobación inicial y el acceso real, comprometiendo la seguridad del programa.

Codificación insegura

Acceso a un archivo verificando primero su existencia con `access()` y luego abriéndolo con `fopen()`. Se pretende mostrar cómo una validación previa separada del uso real del recurso puede ser explotada por una condición de carrera TOCTOU, permitiendo que un atacante cambie el archivo entre ambas operaciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
  //-- Se define la ruta del archivo a validar y abrir
  const char *path = "datos.txt";

  //-- Se verifica si el archivo existe y es accesible
  if (access(path, R_OK) != 0) {
    perror("El archivo no es accesible");
    return EXIT_FAILURE;
  }
  //-- Entre access() y fopen() un atacante podria reemplazar el archivo
  FILE *file = fopen(path, "r");
  if (file == NULL) {
    perror("Error al abrir el archivo");
    return EXIT_FAILURE;
  }
  //-- Se lee y muestra el contenido del archivo
  char buffer[128];
  while (fgets(buffer, sizeof(buffer), file) != NULL) {
    printf("%s", buffer);
  }
  fclose(file);
  return EXIT_SUCCESS;
```

```
}
```

Codificación recomendada

Apertura directa del archivo con `open()` y validación posterior usando `fstat()` sobre el descriptor ya abierto. Se busca demostrar cómo evitar la ventana de tiempo entre la comprobación y el uso, realizando todas las verificaciones sobre el mismo descriptor de archivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
  //-- Se define la ruta del archivo a abrir directamente
  const char *path = "datos.txt";

  //-- Se abre el archivo sin validacion previa separada
  int fd = open(path, O_RDONLY);
  if (fd == -1) {
    perror("Error al abrir el archivo");
    return EXIT_FAILURE;
  }
  //-- Se valida el archivo ya abierto usando su descriptor
  struct stat st;
  if (fstat(fd, &st) == -1) {
    perror("Error al obtener informacion del archivo");
    close(fd);
    return EXIT_FAILURE;
  }
  //-- Se verifica que sea un archivo regular
  if (!S_ISREG(st.st_mode)) {
    fprintf(stderr, "No es un archivo regular\n");
    close(fd);
    return EXIT_FAILURE;
  }
  //-- Se asocia el descriptor a un FILE para facilitar la lectura
  FILE *file = fdopen(fd, "r");
  if (file == NULL) {
    perror("Error al asociar descriptor");
    close(fd);
    return EXIT_FAILURE;
  }
  //-- Se lee y muestra el contenido del archivo
  char buffer[128];
  while (fgets(buffer, sizeof(buffer), file) != NULL) {
```

```
    printf("%s", buffer);
  }
  fclose(file);
  return EXIT_SUCCESS;
}
```

2.10.3.1 Descripción

Las condiciones de carrera TOCTOU (Time-of-Check to Time-of-Use) representan una vulnerabilidad que surge cuando existe un intervalo temporal entre la verificación de una condición sobre un archivo y su posterior uso. Durante este intervalo, el estado del archivo puede cambiar de manera inesperada, invalidando la verificación inicial.

Este tipo de vulnerabilidad ocurre típicamente cuando se realizan operaciones separadas para comprobar propiedades de un archivo (como permisos, existencia o tipo) y posteriormente actuar sobre él. Entre ambas operaciones, un atacante o proceso concurrente puede modificar el archivo, crear enlaces simbólicos maliciosos o alterar las condiciones del sistema de archivos.

Los riesgos asociados incluyen **escalada de privilegios**, acceso no autorizado a archivos sensibles, corrupción de datos y ejecución de código malicioso. Estas vulnerabilidades son especialmente críticas en aplicaciones que manejan archivos con privilegios elevados o en entornos multiusuario.

La mitigación efectiva requiere el uso de operaciones atómicas que combinen la verificación y el uso en una sola llamada al sistema. Las funciones que permiten especificar flags de apertura y verificación simultánea eliminan la ventana temporal vulnerable. Adicionalmente, el uso de descriptores de archivo en lugar de nombres de archivo para operaciones posteriores reduce la exposición a este tipo de ataques.

En sistemas que lo soporten, el empleo de directorios de trabajo seguros y la validación de rutas canónicas proporcionan capas adicionales de protección contra manipulaciones del sistema de archivos durante las operaciones críticas.

Riesgos asociados

- Elevación de privilegios: un atacante puede manipular el archivo entre las operaciones de comprobación y uso para obtener acceso indebido.
- Divulgación de información sensible: se podría acceder, modificar o borrar archivos confidenciales debido a cambio de archivos en el intervalo TOCTOU.
- Corrupción de datos: los archivos podrían ser sobrescritos o alterados maliciosamente antes de su uso real.
- Ejecución de código arbitrario: un archivo manipulado podría contener código dañino que se ejecuta bajo el contexto de la aplicación.

- Violación de integridad: acceso incorrecto puede quebrantar la integridad de la operación esperada, permitiendo resultados inesperados o erróneos.
- Denegación de servicio (DoS): manipulación de archivos durante el intervalo puede causar fallos o comportamientos erráticos que inutilizan la aplicación.

Buenas prácticas

- Utilizar descriptores de archivo con `open()` y banderas seguras en lugar de verificar primero con `access()`.

```
//-- Verificación previa con access() antes de abrir el archivo
if (access("datos.txt", W_OK) == 0) {
  int fd = open("datos.txt", O_WRONLY);
  write(fd, "OK", 2);
  close(fd);
}
```

```
//-- Apertura directa con comprobación atómica del resultado
int fd = open("datos.txt", O_WRONLY);
if (fd != -1) {
  write(fd, "OK", 2);
  close(fd);
}
```

- Crear archivos nuevos usando `O_CREAT | O_EXCL` para evitar sustituciones entre la comprobación y la creación.

```
//-- Comprobación de existencia antes de crear el archivo
if (access("nuevo.txt", F_OK) != 0) {
  int fd = open("nuevo.txt", O_CREAT | O_WRONLY, 0600);
  close(fd);
}
```

```
//-- Creación atómica con O_CREAT | O_EXCL
int fd = open("nuevo.txt", O_CREAT | O_EXCL | O_WRONLY, 0600);
if (fd != -1) {
  close(fd);
}
```

- Evitar seguir enlaces simbólicos utilizando `O_NOFOLLOW` al abrir archivos sensibles.

```
//-- Apertura sin validar si es un enlace simbólico
int fd = open("config.cfg", O_RDONLY);
if (fd != -1) {
  read(fd, buffer, sizeof(buffer));
  close(fd);
```

```
}
```

```
//-- Apertura con O_NOFOLLOW para impedir redirecciones
int fd = open("config.cfg", O_RDONLY | O_NOFOLLOW);
if (fd != -1) {
  read(fd, buffer, sizeof(buffer));
  close(fd);
}
```

- Validar atributos del archivo mediante `fstat()` después de abrirlo en lugar de usar `stat()` previamente.

```
//-- Uso de stat() antes de abrir el archivo
struct stat st;
stat("datos.txt", &st);
if (st.st_uid == getuid()) {
  int fd = open("datos.txt", O_RDONLY);
  close(fd);
}
```

```
//-- Validación del descriptor ya abierto con fstat()
int fd = open("datos.txt", O_RDONLY);
if (fd != -1) {
  struct stat st;
  fstat(fd, &st);
  if (st.st_uid == getuid()) {
    /* uso seguro */
  }
  close(fd);
}
```

2.10.3.2 Estándares metodológicos

- **CERT/C-FIO45**: Avoid TOCTOU race conditions while accessing files.

2.10.4 [c.val.4] - No acceder a un archivo cerrado

ID	c.val.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los archivos cerrados mediante `fclose()` no deben ser utilizados en operaciones posteriores. El acceso a punteros de archivo invalidados puede provocar corrupción de datos o fallos del programa. Esta situación surge frecuentemente cuando se reutilizan

variables de puntero sin verificar su estado o cuando múltiples funciones operan sobre el mismo archivo.

Codificación insegura

Gestión básica de archivos donde se abre un fichero, se cierra y posteriormente se intenta volver a usar el mismo puntero. Se busca observar el comportamiento cuando se accede a un recurso que ya fue liberado y cómo esto puede provocar fallos de ejecución o comportamientos indefinidos.

```
#include <stdio.h>

//-- Demostración de uso incorrecto de un archivo cerrado
int main() {
  FILE *file = fopen("datos.txt", "w");
  if (file == NULL) {
    perror("Error al abrir el archivo");
    return 1;
  }
  //-- Escritura inicial en el archivo
  fprintf(file, "Linea inicial\n");

  //-- Cierre del archivo
  fclose(file);

  //-- Intento de uso del archivo después de cerrarlo (comportamiento indefinido)
  fprintf(file, "Segunda linea\n");

  return 0;
}
```

Codificación recomendada

Gestión segura de archivos donde se valida la apertura, se utiliza el recurso mientras está disponible y se evita cualquier acceso posterior tras su cierre, incluyendo la invalidación explícita del puntero para prevenir usos accidentales.

```
#include <stdio.h>

//-- Demostración de uso seguro de un archivo
int main() {
  FILE *file = fopen("datos.txt", "w");
  if (file == NULL) {
    perror("Error al abrir el archivo");
    return 1;
  }
  //-- Escritura válida en el archivo abierto
  if (fprintf(file, "Linea segura\n") < 0) {
```

```
    perror("Error al escribir en el archivo");
    fclose(file);
    return 1;
  }
  //-- Cierre controlado del archivo
  if (fclose(file) != 0) {
    perror("Error al cerrar el archivo");
    return 1;
  }
  //-- Invalida el puntero para evitar usos posteriores accidentales
  file = NULL;

  //-- Validación antes de cualquier uso futuro
  if (file != NULL) {
    fprintf(file, "No se ejecutara\n");
  }
  return 0;
}
```

2.10.4.1 Descripción

Esta regla establece que no se debe intentar realizar operaciones de lectura, escritura o cualquier otra manipulación sobre un archivo que ha sido previamente cerrado mediante `fclose()` o que nunca fue abierto correctamente.

El acceso a archivos cerrados constituye un **comportamiento indefinido** que puede manifestarse de múltiples formas impredecibles. El sistema operativo puede haber liberado los recursos asociados al descriptor de archivo, reasignado el identificador a otro archivo, o marcado la estructura de datos como inválida.

Los riesgos asociados incluyen corrupción de datos, fallos de segmentación, comportamiento errático del programa y potenciales vulnerabilidades de seguridad. En algunos casos, el programa puede parecer funcionar correctamente mientras opera sobre datos incorrectos o recursos no válidos.

La mitigación efectiva requiere un **control riguroso del estado de los archivos** a lo largo del ciclo de vida del programa. Esto implica verificar el éxito de las operaciones de apertura, mantener un seguimiento claro del estado de cada archivo, y establecer patrones de código que eviten el uso accidental de punteros a archivos cerrados.

Las técnicas de mitigación incluyen la inicialización de punteros de archivo a `NULL` tras el cierre, la implementación de verificaciones de estado antes de cada operación, y el diseño de estructuras de control que garanticen la correspondencia entre operaciones de apertura y cierre.

Riesgos asociados

- Lectura o escritura de datos inválidos o erróneos.

- Comportamiento indefinido de la aplicación.
- Corrupción de memoria (buffer overflows, uso de memoria no inicializada, etc.).
- Errores de segmentación (segmentation fault) y fallos en tiempo de ejecución.
- Posibles condiciones de denegación de servicio (DoS) por el cierre inesperado del programa.
- Pérdida o corrupción de archivos y datos persistentes.
- Exposición de información sensible en registros de errores o volcados de memoria.
- Dificultad en la trazabilidad y depuración del código debido a errores no controlados.

Buenas prácticas

- Validar el estado del archivo antes de utilizar el descriptor o puntero.

```
//-- Uso de puntero a archivo después de cerrarlo
FILE *f = fopen("datos.txt", "r");
fclose(f);
char buffer[32];
fgets(buffer, sizeof(buffer), f);
```

```
//-- Validación del puntero y control del ciclo de vida antes de usarlo
FILE *f2 = fopen("datos.txt", "r");
if (f2 != NULL) {
  char buffer2[32];
  fgets(buffer2, sizeof(buffer2), f2);
  fclose(f2);
  f2 = NULL;
}
```

- Asignar `NULL` al puntero después de cerrar el archivo para evitar reutilización accidental.

```
//-- Cierre de archivo sin invalidar el puntero
FILE *log = fopen("log.txt", "w");
if (log != NULL) {
  fclose(log);
}
fprintf(log, "Evento\n");
```

```
//-- Invalidación explícita del puntero tras el cierre
FILE *log2 = fopen("log.txt", "w");
if (log2 != NULL) {
```

```
  fclose(log2);
  log2 = NULL;
}
if (log2 != NULL) {
  fprintf(log2, "Evento\n");
}
```

- Encapsular la gestión de archivos para controlar apertura, uso y cierre en un único flujo seguro.

```
//-- Uso del archivo fuera del bloque donde fue gestionado
FILE *cfg;
void abrir() {
  cfg = fopen("config.cfg", "r");
  fclose(cfg);
}
void leer() {
  char linea[64];
  fgets(linea, sizeof(linea), cfg);
}
```

```
//-- Encapsulamiento del ciclo completo de vida del archivo
void procesar_config() {
  FILE *cfg2 = fopen("config.cfg", "r");
  if (cfg2 == NULL) return;
  char linea2[64];
  fgets(linea2, sizeof(linea2), cfg2);
  fclose(cfg2);
}
```

2.10.4.2 Estándares metodológicos

- **CERT/C-FIO46**: Do not access a closed file.

2.10.5 [c.val.5] - Usar únicamente cadenas de formato válidas

ID	c.val.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las funciones de la familia `printf` y `scanf` requieren que las cadenas de formato coincidan exactamente con el tipo y número de argumentos proporcionados. **Los desajustes entre especificadores de formato y argumentos** pueden provocar corrupción de memoria o vulnerabilidades de seguridad explotables por atacantes.

Codificación insegura

Se muestra el uso de funciones de salida formateada donde la cadena de formato proviene de datos externos. La intención es entender cómo el uso inadecuado de cadenas de formato puede provocar vulnerabilidades como ataques de formato, y qué comportamiento se obtiene al compilar y ejecutar el programa.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char input[100];

  //-- Solicita al usuario una cadena que será usada como formato
  printf("Introduce un texto: ");
  fgets(input, sizeof(input), stdin);

  //-- Elimina el salto de línea si existe
  input[strcspn(input, "\n")] = '\0';

  //-- Uso directo de la entrada como cadena de formato
  //-- Si el usuario introduce especificadores como %x o %s se produce
comportamiento inesperado
  printf(input);
  printf("\n");

  return 0;
}
```

Codificación recomendada

Se presenta el uso seguro de funciones de salida formateada empleando una cadena de formato constante y válida. El objetivo es garantizar que los datos externos se traten únicamente como argumentos y no como formato ejecutable, evitando vulnerabilidades de tipo format string.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char input[100];

  //-- Solicita al usuario una cadena que será mostrada por pantalla
  printf("Introduce un texto: ");
  fgets(input, sizeof(input), stdin);

  //-- Elimina el salto de línea si existe
  input[strcspn(input, "\n")] = '\0';
```

```
  //-- Uso de una cadena de formato fija y válida
  //-- El contenido del usuario se pasa como argumento y no como formato
  printf("%s\n", input);

  return 0;
}
```

2.10.5.1 Descripción

Las funciones de la familia `printf` y `scanf` utilizan cadenas de formato para especificar cómo interpretar y procesar los argumentos de entrada o salida. Estas cadenas contienen especificadores de formato que deben coincidir exactamente con el tipo y número de argumentos proporcionados.

El uso de cadenas de formato inválidas o inconsistentes con los argumentos puede provocar **comportamiento indefinido** en tiempo de ejecución. Los especificadores de formato incorrectos pueden causar que el programa lea o escriba memoria de forma impredecible, generando corrupción de datos, fallos de segmentación o vulnerabilidades de seguridad.

Los riesgos se materializan cuando existe una discrepancia entre los especificadores de formato y los tipos de datos reales de los argumentos. Esta situación puede derivar en **lectura o escritura de memoria no autorizada**, permitiendo potencialmente la ejecución de código malicioso o la filtración de información sensible.

La validación de las cadenas de formato debe realizarse tanto en **tiempo de compilación como en tiempo de ejecución** cuando sea posible. Los compiladores modernos proporcionan advertencias cuando detectan inconsistencias entre especificadores de formato y argumentos, facilitando la identificación temprana de estos problemas.

La mitigación efectiva incluye el uso de herramientas de análisis estático que verifiquen la correspondencia entre formatos y argumentos, así como la implementación de controles adicionales cuando se manejan cadenas de formato dinámicas o proporcionadas por fuentes externas.

Riesgos asociados

- Ejecución de código arbitrario debido a ataques de formato de cadena (format string attack).
- Revelación de información sensible o confidencial almacenada en la pila o memoria.
- Corrupción de memoria y posibles vulneraciones a la integridad de los datos.
- Desbordamiento de buffer causado por un manejo incorrecto de los argumentos de formato.

- Comportamiento indefinido que puede provocar fallas inesperadas o bloqueos del programa.
- Apertura de vectores para ataques de escalada de privilegios.
- Dificultad para detectar y depurar errores de seguridad.

Buenas prácticas

- Validar que las cadenas de formato sean constantes y no provengan directamente de entrada externa.

```
//-- Uso directo de entrada externa como formato
char buffer[128];
scanf("%127s", buffer);
printf(buffer);
```

```
//-- Uso de formato constante controlado
char buffer[128];
scanf("%127s", buffer);
printf("%s", buffer);
```

- Evitar especificadores de formato incompatibles con el tipo real de las variables.

```
//-- Especificador incompatible con el tipo real
int valor = 10;
printf("%f", valor);
```

```
//-- Especificador coherente con el tipo declarado
int valor = 10;
printf("%d", valor);
```

- Limitar explícitamente el tamaño en formatos de entrada para prevenir desbordamientos.

```
//-- Lectura sin límite de tamaño en la cadena de formato
char nombre[16];
scanf("%s", nombre);
```

```
//-- Lectura con límite acorde al tamaño del buffer
char nombre[16];
scanf("%15s", nombre);
```

- Revisar y validar dinámicamente las cadenas de formato cuando se construyan de manera programática.

```
//-- Construcción dinámica sin validación del formato
char formato[32] = "%s %n";
printf(formato, "texto");
```

```
//-- Construcción dinámica restringiendo especificadores permitidos
char formato[32] = "%s %d";
printf(formato, "texto", 5);
```

2.10.5.2 Estándares metodológicos

- **CERT/C-FIO47**: Use valid format strings.

2.10.6 [c.val.6] - Considerar siempre la posibilidad de entradas malformadas al leer datos

ID	c.val.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los programas que procesan datos de entrada sin validación previa quedan expuestos a **vulnerabilidades de seguridad**. Las entradas malformadas pueden provocar desbordamientos de buffer, corrupción de memoria, denegación de servicio o ejecución de código arbitrario. Esta exposición se agrava cuando se confía implícitamente en el formato de los datos recibidos.

Codificación insegura

Lectura de un número entero desde la entrada estándar sin validar el contenido recibido. Se pretende mostrar cómo una entrada malformada puede provocar comportamientos inesperados cuando no se comprueba el resultado de la lectura ni el formato real de los datos.

```
#include <stdio.h>

//-- Programa que lee un entero sin validar errores de entrada
int main() {
  int numero;

  //-- Solicita un número al usuario
  printf("Introduce un numero entero: ");
  scanf("%d", &numero);

  //-- Usa el valor sin comprobar si la lectura fue correcta
  printf("El numero multiplicado por 2 es: %d\n", numero * 2);
```

```
  //-- Fin del programa
  return 0;
}
```

Codificación recomendada

Lectura segura de un número entero validando que la entrada no esté malformada. Se comprueba que la lectura sea correcta, que no existan caracteres adicionales no esperados y que la conversión sea válida antes de utilizar el dato.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

//-- Programa que valida correctamente la entrada antes de usarla
int main() {
  char buffer[64];
  char *endptr;
  long numero;

  //-- Solicita un número al usuario
  printf("Introduce un numero entero: ");

  //-- Lee la linea completa de forma segura
  if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
    //-- Error al leer la entrada
    fprintf(stderr, "Error al leer la entrada.\n");
    return 1;
  }
  //-- Reinicia errno antes de la conversion
  errno = 0;
  numero = strtol(buffer, &endptr, 10);

  //-- Verifica errores de conversion o entrada malformada
  if (errno != 0 || endptr == buffer || (*endptr != '\n' && *endptr != '\0')) {
    //-- Entrada no valida
    fprintf(stderr, "Entrada no valida. Debe ser un numero entero.\n");
    return 1;
  }
  //-- Usa el valor solo despues de validarlo
  printf("El numero multiplicado por 2 es: %ld\n", numero * 2);

  //-- Fin del programa
  return 0;
}
```

2.10.6.1 Descripción

Esta regla establece que toda operación de lectura de datos debe implementar validación previa para detectar y manejar entradas que no cumplan con el formato esperado. La validación debe verificar tanto la estructura como el contenido de los datos antes de procesarlos.

Las entradas malformadas representan una superficie de ataque significativa, ya que pueden provocar comportamientos impredecibles en el programa, corrupción de memoria o permitir la ejecución de código malicioso. Los atacantes frecuentemente explotan la ausencia de validación para introducir datos diseñados específicamente para comprometer la seguridad del sistema.

La validación debe aplicarse a todas las fuentes de entrada: datos introducidos por usuarios, archivos, comunicaciones de red, argumentos de línea de comandos, variables de entorno y cualquier información externa al programa. Ninguna fuente debe considerarse inherentemente confiable.

Los mecanismos de validación deben definir claramente qué constituye una entrada válida, estableciendo límites de longitud, rangos de valores permitidos, caracteres aceptables y estructura esperada. La validación debe realizarse utilizando listas de elementos permitidos en lugar de listas de elementos prohibidos, ya que este enfoque ofrece mayor seguridad.

Cuando se detecta una entrada malformada, el programa debe manejar la situación de forma controlada: rechazar la entrada, registrar el evento si es necesario, y continuar la ejecución de manera segura sin procesar los datos inválidos. El manejo de errores debe ser explícito y no debe revelar información sensible sobre la estructura interna del programa.

La implementación efectiva de esta regla requiere el uso de funciones de entrada seguras que permitan especificar límites y validar formatos, así como la implementación de rutinas de validación personalizadas cuando las funciones estándar no proporcionan la granularidad necesaria.

Riesgos asociados

- Desbordamiento de buffer debido a entradas más largas de lo esperado.
- Inyección de código o comandos maliciosos a través de datos manipulados por el atacante.
- Corrupción de memoria o variables por no validar formatos y límites de entrada.
- Denegación de servicio (DoS) causada por la explotación de casos límite o errores de parseo.
- Lectura fuera de límites de memoria del programa (out-of-bounds read).
- Fugas de información sensible por procesar datos inesperados.

- Comportamiento indefinido del programa, incluyendo posibles caídas o bloqueos.
- Ataques de escalada de privilegios si las entradas malformadas afectan controles de acceso.
- Ejecución de instrucciones no previstas, generando condiciones de carrera o corrupción de estado.
- Vulnerabilidades a ataques de formato (como `%n` en `printf`) si no se controla la entrada del usuario.

Buenas prácticas

- Validar siempre el resultado de funciones de entrada antes de usar los datos leídos.

```
//-- Uso directo del valor leído sin verificar resultado
int edad;
scanf("%d", &edad);
printf("Edad: %d", edad);
```

```
//-- Verificación del valor de retorno de scanf antes de usar la variable
int edad;
if (scanf("%d", &edad) == 1) {
  printf("Edad: %d", edad);
} else {
  fprintf(stderr, "Entrada inválida");
}
```

- Limitar el tamaño de los datos leídos para evitar desbordamientos de buffer.

```
//-- Lectura sin límite de tamaño en buffer fijo
char nombre[10];
scanf("%s", nombre);
```

```
//-- Especificación de ancho máximo en la cadena de formato
char nombre[10];
scanf("%9s", nombre);
```

- Validar rangos y límites después de convertir datos numéricos.

```
//-- Conversión sin comprobación de rango
char input[20];
fgets(input, sizeof(input), stdin);
int valor = atoi(input);
```

```
//-- Conversión con verificación de errores y límites
char input[20];
fgets(input, sizeof(input), stdin);
char *endptr;
long valor = strtol(input, &endptr, 10);
if (endptr == input || *endptr != '\n' || valor < 0 || valor > 100) {
  fprintf(stderr, "Número fuera de rango");
}
```

- Comprobar que la entrada no contenga caracteres inesperados antes de procesarla.

```
//-- Uso de la entrada sin validar caracteres permitidos
char codigo[8];
fgets(codigo, sizeof(codigo), stdin);
procesar_codigo(codigo);
```

```
//-- Validación de caracteres alfanuméricos antes de procesar
char codigo[8];
fgets(codigo, sizeof(codigo), stdin);
int valido = 1;
for (int i = 0; codigo[i] != '\0' && codigo[i] != '\n'; i++) {
  if (!isalnum((unsigned char)codigo[i])) {
    valido = 0;
    break;
  }
}
if (valido) {
  procesar_codigo(codigo);
} else {
  fprintf(stderr, "Formato inválido");
}
```

- Inicializar variables antes de usarlas cuando la lectura pueda fallar.

```
//-- Variable potencialmente no inicializada si falla la lectura
int cantidad;
if (scanf("%d", &cantidad) != 1) {
  fprintf(stderr, "Error de lectura");
}
printf("%d", cantidad);
```

```
//-- Inicialización previa y uso condicionado al éxito de lectura
int cantidad = 0;
if (scanf("%d", &cantidad) == 1) {
  printf("%d", cantidad);
} else {
```

```
  fprintf(stderr, "Error de lectura");
}
```

2.10.6.2 Estándares metodológicos

- **MISRA C-20.9**: The input/output library <stdio.h> shall not be used in production code.
- **CERT/C-FIO30**: Exclude user input from format strings.
- **CERT/C-FIO47**: Use valid format strings.

2.10.7 [c.val.7] - Garantizar que los tamaños de arrays de longitud variable estén dentro de rangos válidos

ID	c.val.7	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los arrays de longitud variable permiten declarar arrays cuyo tamaño se determina en tiempo de ejecución, pero pueden generar **desbordamientos de pila** o **consumo excesivo de memoria** cuando se especifican tamaños inadecuados. Los valores negativos, extremadamente grandes o no validados pueden provocar fallos del programa o vulnerabilidades de seguridad.

Codificación insegura

Uso de un array de longitud variable (VLA) cuyo tamaño se obtiene desde la entrada del usuario. Se pretende mostrar cómo un tamaño no validado puede provocar desbordamientos de pila o comportamiento indefinido si el valor es negativo o excesivamente grande.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Solicita al usuario el tamaño del array
  int n;
  printf("Ingrese el tamaño del array: ");
  scanf("%d", &n);

  //-- Se declara un VLA sin validar el rango
  int datos[n];

  //-- Inicializa y muestra los valores
  for (int i = 0; i < n; i++) {
```

```
    datos[i] = i;
    printf("datos[%d] = %d\n", i, datos[i]);
  }
  return 0;
}
```

Codificación recomendada

Uso de un array de longitud variable (VLA) validando previamente que el tamaño introducido se encuentre dentro de un rango seguro y razonable. Se busca evitar tamaños negativos, cero o excesivamente grandes que puedan causar fallos o consumo indebido de memoria.

```
#include <stdio.h>
#include <stdlib.h>

#define MIN_SIZE 1
#define MAX_SIZE 1000

int main(void) {
  //-- Solicita al usuario el tamaño del array
  int n;
  printf("Ingrese el tamaño del array (1-1000): ");
  if (scanf("%d", &n) != 1) {
    //-- Error en la lectura de datos
    fprintf(stderr, "Entrada invalida\n");
    return 1;
  }
  //-- Valida que el tamaño esté dentro de un rango seguro
  if (n < MIN_SIZE || n > MAX_SIZE) {
    //-- Tamaño fuera de los limites permitidos
    fprintf(stderr, "El tamaño debe estar entre %d y %d\n", MIN_SIZE, MAX_SIZE);
    return 1;
  }
  //-- Se declara el VLA solo después de validar el tamaño
  int datos[n];

  //-- Inicializa y muestra los valores
  for (int i = 0; i < n; i++) {
    datos[i] = i;
    printf("datos[%d] = %d\n", i, datos[i]);
  }
  return 0;
}
```

2.10.7.1 Descripción

Los arrays de longitud variable permiten declarar arrays cuyo tamaño se determina en tiempo de ejecución mediante una expresión. Esta flexibilidad introduce riesgos de seguridad cuando el valor que determina el tamaño no se valida adecuadamente antes de la declaración del array.

El problema surge porque estos arrays se almacenan típicamente en el stack, que tiene un espacio limitado. Si el tamaño especificado es excesivamente grande, negativo o proviene de datos no confiables, puede provocar **desbordamientos del stack**, **consumo excesivo de memoria**, o **comportamiento indefinido** del programa.

Los valores de tamaño pueden originarse de diversas fuentes: entrada del usuario, archivos de configuración, parámetros de red o cálculos aritméticos. Cuando estos valores no se verifican, un atacante puede manipularlos para comprometer la estabilidad y seguridad de la aplicación.

La validación debe considerar tanto **límites superiores** como **límites inferiores**. Los valores negativos pueden interpretarse como números muy grandes debido a la conversión de tipos, mientras que valores excesivamente grandes pueden agotar los recursos disponibles del sistema.

Para mitigar estos riesgos, es necesario implementar **validación explícita** de los valores de tamaño antes de utilizarlos en la declaración de arrays. Esta validación debe establecer rangos apropiados basados en los recursos disponibles y los requisitos específicos de la aplicación.

Las estrategias de mitigación incluyen el uso de **constantes predefinidas** para límites máximos, **verificación de rangos** mediante condiciones, y el manejo adecuado de casos donde la validación falla. En situaciones donde se requiere flexibilidad en el tamaño, considerar alternativas como **asignación dinámica de memoria** con las correspondientes verificaciones de éxito en la asignación.

Riesgos asociados

- Desbordamiento de buffer (buffer overflow) al declarar arrays de longitud variable con tamaños no validados.
- Lectura/escritura fuera de los límites del array, provocando corrupción de memoria.
- Exposición de datos sensibles debido a accesos fuera de los límites del array.
- Condiciones de denegación de servicio (DoS) por fallos de ejecución inesperados.
- Comportamiento indefinido, que puede derivar en fallos del programa o vulnerabilidades explotables.
- Ataques de ejecución de código arbitrario a través de la explotación de los desbordamientos.

- Errores de segmentación (segmentation faults) que pueden dar lugar a caídas de la aplicación.
- Difícil detección y depuración de errores, lo que incrementa la posibilidad de mantener código vulnerable.
- Incremento en la superficie de ataque al sistema o aplicación al dejar abiertas posibles vulnerabilidades de memoria.

Buenas prácticas

- Validar explícitamente que el tamaño del array de longitud variable (VLA) sea mayor que cero y menor que un límite máximo definido.

```
//-- Tamaño sin validación de rango
void procesar(int n) {
  int buffer[n];
  buffer[0] = 0;
}
```

```
//-- Validación de límites antes de declarar el VLA
#define MAX_BUFFER 1024
void procesar(int n) {
  if (n <= 0 || n > MAX_BUFFER) {
    return;
  }
  int buffer[n];
  buffer[0] = 0;
}
```

- Comprobar que el tamaño no provoque desbordamiento al calcular memoria derivada (por ejemplo, multiplicaciones).

```
//-- Cálculo de tamaño sin verificar overflow
void crear(int n) {
  int matriz[n * 4];
  matriz[0] = 1;
}
```

```
//-- Verificación previa de overflow antes de usar el tamaño
#include <limits.h>
void crear(int n) {
  if (n <= 0 || n > INT_MAX / 4) {
    return;
  }
  int matriz[n * 4];
  matriz[0] = 1;
```

```
}
```

- Evitar el uso de VLAs cuando el tamaño proviene de entrada externa no confiable; preferir memoria dinámica con validación previa.

```
//-- Uso directo de entrada externa como tamaño de VLA
void leer() {
  int n;
  scanf("%d", &n);
  char datos[n];
  datos[0] = 'A';
}
```

```
//-- Validación de entrada y uso de memoria dinámica controlada
#define MAX_DATOS 2048
void leer() {
  int n;
  if (scanf("%d", &n) != 1 || n <= 0 || n > MAX_DATOS) {
    return;
  }
  char *datos = malloc(n);
  if (datos == NULL) {
    return;
  }
  datos[0] = 'A';
  free(datos);
}
```

- Definir políticas de límites coherentes y reutilizables mediante constantes simbólicas para evitar valores arbitrarios.

```
//-- Uso de valor arbitrario sin política definida
void funcion(int n) {
  if (n > 5000) return;
  int array[n];
  array[0] = 10;
}
```

```
//-- Uso de constante simbólica centralizada para el límite
#define LIMITE_ARREGLO 1024
void funcion(int n) {
  if (n <= 0 || n > LIMITE_ARREGLO) {
    return;
  }
  int array[n];
  array[0] = 10;
```

```
}
```

2.10.7.2 Estándares metodológicos

- **CERT/C-ARR32**: Ensure size arguments for variable length arrays are in a valid range.

2.11 Gestión de errores

2.11.1 [c.err.1] - Detectar y manejar errores de bibliotecas estándar

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

Las funciones de las bibliotecas estándar pueden fallar durante la ejecución y reportar errores a través de diversos mecanismos como valores de retorno, variables globales o estados internos. La ausencia de verificación de estos errores puede provocar **corrupción de datos** o **vulnerabilidades de seguridad** cuando el programa continúa ejecutándose bajo condiciones de fallo no detectadas.

Codificación insegura

Se muestra un programa que abre un archivo y convierte un texto a número utilizando funciones de la biblioteca estándar. El objetivo es observar qué ocurre cuando no se validan los valores de retorno ni se gestionan las condiciones de error que pueden producirse en tiempo de ejecución.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  //-- Se intenta abrir un archivo sin verificar si fopen falla
  FILE *file = fopen("datos.txt", "r");

  //-- Se usa el puntero directamente sin comprobar si es NULL
  char buffer[64];
  fgets(buffer, sizeof(buffer), file);

  //-- Se convierte el contenido a entero sin validar errores
  int valor = atoi(buffer);

  //-- Se muestra el valor sin saber si la conversión fue válida
  printf("Valor leido: %d\n", valor);

  //-- Se cierra el archivo sin comprobar si estaba correctamente abierto
```

```
  fclose(file);

  return 0;
}
```

Codificación recomendada

Se presenta un programa similar que abre un archivo y convierte un texto a número, incorporando validaciones de errores en cada llamada relevante de la biblioteca estándar. El objetivo es aprender a comprobar valores de retorno, usar errno cuando corresponde y evitar comportamientos indefinidos.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main(void) {
  //-- Se intenta abrir el archivo y se valida el resultado
  FILE *file = fopen("datos.txt", "r");
  if (file == NULL) {
    //-- Se informa del error usando perror
    perror("Error al abrir el archivo");
    return EXIT_FAILURE;
  }
  //-- Se lee una linea y se valida el resultado de fgets
  char buffer[64];
  if (fgets(buffer, sizeof(buffer), file) == NULL) {
    //-- Se verifica si fue error o fin de archivo
    if (ferror(file)) {
      perror("Error al leer el archivo");
    } else {
      printf("No hay datos en el archivo\n");
    }
    fclose(file);
    return EXIT_FAILURE;
  }
  //-- Se convierte el texto a entero usando strtol con validacion
  errno = 0;
  char *endptr;
  long valor = strtol(buffer, &endptr, 10);

  if (errno != 0) {
    //-- Se detecta error de rango u otro error de conversion
    perror("Error en la conversion");
    fclose(file);
    return EXIT_FAILURE;
  }
```

```
  if (endptr == buffer) {
    //-- No se encontraron digitos validos
    printf("Entrada no valida\n");
    fclose(file);
    return EXIT_FAILURE;
  }
  //-- Uso seguro del valor convertido
  printf("Valor leido: %ld\n", valor);

  fclose(file);
  return EXIT_SUCCESS;
}
```

2.11.1.1 Descripción

Las funciones de las bibliotecas estándar pueden fallar durante su ejecución por diversas razones, como recursos insuficientes, parámetros inválidos o condiciones del sistema. Muchas de estas funciones comunican el estado de la operación a través de valores de retorno específicos o mediante la modificación de variables globales como `errno`.

La omisión en la verificación de estos indicadores de error genera **comportamientos impredecibles** en el programa. Las operaciones que aparentan ejecutarse correctamente pueden haber fallado silenciosamente, dejando el programa en un estado inconsistente. Esta situación compromete tanto la **integridad de los datos** como la **estabilidad del sistema**.

Los errores no detectados se propagan a través del flujo de ejecución, amplificando su impacto. Una función de asignación de memoria que falla sin ser detectada puede provocar accesos a punteros nulos. Una operación de entrada/salida fallida puede resultar en el procesamiento de datos corruptos o incompletos.

La detección sistemática de errores requiere **verificar los valores de retorno** de todas las funciones que pueden fallar y **consultar las variables de estado** cuando corresponda. El manejo adecuado implica implementar estrategias de recuperación apropiadas para cada tipo de error, que pueden incluir la liberación de recursos, el registro del error y la terminación controlada del programa.

Las funciones críticas para la seguridad, como las relacionadas con criptografía, autenticación o validación de entrada, requieren una verificación especialmente rigurosa de sus códigos de error, ya que los fallos en estas operaciones pueden comprometer directamente la seguridad del sistema.

Riesgos asociados

- Fallos silenciosos en la ejecución del programa al no detectar fallos de función.
- Corrupción de memoria por uso de resultados inválidos de funciones fallidas.

- Fuga de información confidencial al continuar la ejecución ante errores no manejados.
- Condiciones de carrera o estados inconsistentes por falta de restauración de contexto ante errores.
- Denegación de servicio (DoS) causada por bloqueos o ciclos infinitos al no gestionar correctamente condiciones de error.
- Vulnerabilidades explotables (por ejemplo, buffer overflows o desbordamiento de enteros) al no validar los retornos de funciones críticas como `malloc`, `fread`, `snprintf`, etc.
- Pérdida de datos o corrupción de archivos debido a la omisión de manejo de errores de E/S de archivos.
- Dificultad en la detección y depuración de problemas por ausencia de logs o reportes de error.

Buenas prácticas

- Verificar siempre el valor de retorno de funciones de entrada/salida antes de usar los datos obtenidos.

```
//-- Uso de fread sin validar cantidad leída
FILE *f = fopen("datos.bin", "rb");
char buffer[100];
fread(buffer, 1, sizeof(buffer), f);
fclose(f);
```

```
//-- Validación de apertura y cantidad leída antes de procesar
FILE *f = fopen("datos.bin", "rb");
if (f != NULL) {
  char buffer[100];
  size_t leidos = fread(buffer, 1, sizeof(buffer), f);
  if (leidos == sizeof(buffer)) {
    procesar(buffer);
  }
  fclose(f);
}
```

- Comprobar errores en funciones de asignación dinámica de memoria antes de utilizar el puntero devuelto.

```
//-- Uso de malloc sin comprobar resultado
char *ptr = malloc(256);
strcpy(ptr, "texto");
free(ptr);
```

```
//-- Validación del puntero antes de su uso
char *ptr = malloc(256);
if (ptr != NULL) {
  strncpy(ptr, "texto", 255);
  ptr[255] = '\0';
  free(ptr);
}
```

- Evaluar y manejar errores en conversiones y funciones que establecen `errno`.

```
//-- Conversión con strtol sin comprobar errores
char *entrada = "99999999999999999999";
long valor = strtol(entrada, NULL, 10);
usar_valor(valor);
```

```
//-- Reinicio y verificación de errno y rango
char *entrada = "99999999999999999999";
errno = 0;
char *fin = NULL;
long valor = strtol(entrada, &fin, 10);
if (errno == 0 && fin != entrada && *fin == '\0') {
  usar_valor(valor);
}
```

2.11.1.2 Estándares metodológicos

- **CERT/C-ERR33**: Detect and handle standard library errors.

2.11.2 [c.err.2] - Detectar errores al convertir cadenas a números

ID	c.err.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La conversión de cadenas a valores numéricos mediante funciones como `atoi()`, `strtol()` o `sscanf()` puede generar **vulnerabilidades** cuando no se validan adecuadamente los datos de entrada. Las cadenas malformadas, los valores fuera de rango o los caracteres no numéricos pueden provocar **conversiones erróneas silenciosas** que comprometen la lógica del programa y crean vulnerabilidades de seguridad.

Codificación insegura

Conversión de una cadena recibida por línea de comandos a un número entero para realizar una operación aritmética. Se pretende mostrar cómo se maneja la validación de errores cuando el valor introducido no es numérico o está fuera de rango.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  //-- Se verifica que se haya proporcionado un argumento
  if (argc != 2) {
    printf("Uso: %s <numero>\n", argv[0]);
    return 1;
  }
  //-- Conversion directa sin validacion de errores
  int valor = atoi(argv[1]);

  //-- Se utiliza el valor convertido sin comprobar si la cadena era valida
  printf("El doble es: %d\n", valor * 2);

  return 0;
}
```

Codificación recomendada

Conversión de una cadena recibida por línea de comandos a un número entero validando posibles errores como entrada no numérica o desbordamiento. Se pretende aprender a detectar fallos de conversión usando funciones seguras y comprobaciones explícitas.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

int main(int argc, char *argv[]) {
  //-- Se verifica que se haya proporcionado un argumento
  if (argc != 2) {
    printf("Uso: %s <numero>\n", argv[0]);
    return 1;
  }
  //-- Preparacion para detectar errores de conversion
  char *endptr;
  errno = 0;

  //-- Conversion usando strtol con control de errores
  long valor = strtol(argv[1], &endptr, 10);
```

```
  //-- Validacion de errores: sin digitos, caracteres extra o desbordamiento
  if (endptr == argv[1] || *endptr != '\0') {
    printf("Error: la entrada no es un numero valido\n");
    return 1;
  }
  if ((errno == ERANGE) || (valor > INT_MAX) || (valor < INT_MIN)) {
    printf("Error: numero fuera de rango\n");
    return 1;
  }
  //-- Uso seguro del valor validado
  printf("El doble es: %ld\n", valor * 2);

  return 0;
}
```

2.11.2.1 Descripción

La conversión de cadenas de caracteres a valores numéricos representa una operación común que requiere validación rigurosa para garantizar la integridad de los datos procesados. Las funciones estándar de conversión pueden fallar por múltiples razones: formato incorrecto de la cadena de entrada, valores fuera del rango representable del tipo de destino, presencia de caracteres no numéricos o cadenas vacías.

El problema principal radica en que muchas funciones de conversión no proporcionan mecanismos claros para distinguir entre una conversión exitosa y un fallo. Algunas retornan valores específicos en caso de error, mientras que otras modifican variables globales como `errno` para indicar condiciones de error. Esta inconsistencia en el manejo de errores puede llevar a que los fallos de conversión pasen desapercibidos.

Los riesgos asociados incluyen el procesamiento de datos incorrectos que pueden alterar la lógica del programa, causar comportamientos impredecibles, o generar vulnerabilidades de seguridad. La falta de detección de errores puede resultar en el uso de valores no inicializados o valores por defecto inesperados que comprometan la funcionalidad del sistema.

La mitigación efectiva requiere la implementación de verificaciones exhaustivas después de cada operación de conversión. Esto incluye la validación del valor de retorno de las funciones de conversión, la verificación del estado de `errno` cuando sea aplicable, y la confirmación de que toda la cadena de entrada fue procesada correctamente. También resulta necesario establecer rangos válidos para los valores convertidos y rechazar aquellos que excedan los límites esperados para el contexto específico de la aplicación.

Riesgos asociados

- Desbordamientos de buffer por entrada inesperadamente grande o malformada.

- Desbordamiento o subdesbordamiento de variables numéricas al no validar rangos.
- Lectura y procesamiento de valores inválidos (garbage values) en la aplicación.
- Comportamiento indefinido al operar con valores incorrectos.
- Vulnerabilidades de seguridad como denegación de servicio (DoS) si se procesan datos inesperados.
- Errores de lógica en la aplicación por procesamiento incorrecto de datos de entrada.
- Fallo en controles de acceso o validaciones si las conversiones fallidas no se detectan.
- Vulnerabilidad a ataques de manipulación de entrada, como Integer Overflow o Numeric Truncation.
- Filtración involuntaria de información sensible mediante errores.

Buenas prácticas

- Validar errores usando `strtol` en lugar de `atoi` y comprobar `endptr`.

```
//-- Uso de atoi sin validar errores
int valor = atoi(cadena);
```

```
//-- Uso de strtol verificando conversion completa
char *endptr;
long valor = strtol(cadena, &endptr, 10);
if (endptr == cadena || *endptr != '\0') {
  /* manejar error */
}
```

- Comprobar desbordamientos y subdesbordamientos mediante `errno`.

```
//-- Conversion sin verificar limites del tipo
long numero = strtol(cadena, NULL, 10);
```

```
//-- Verificacion de rango usando errno
errno = 0;
long numero = strtol(cadena, NULL, 10);
if (errno == ERANGE) {
  /* manejar desbordamiento o subdesbordamiento */
}
```

- Validar que la cadena de entrada no sea nula antes de convertir.

```
//-- Conversion sin validar puntero de entrada
long n = strtol(cadena, NULL, 10);
```

```
//-- Validacion previa del puntero antes de convertir
if (cadena == NULL) {
  /* manejar puntero nulo */
}
long n = strtol(cadena, NULL, 10);
```

- Verificar que el valor convertido esté dentro del rango esperado del tipo destino.

```
//-- Asignacion directa a int sin validar rango
int edad = (int)strtol(cadena, NULL, 10);
```

```
//-- Validacion explicita del rango antes de asignar
long temp = strtol(cadena, NULL, 10);
if (temp < 0 || temp > 120) {
  /* manejar valor fuera de rango permitido */
}
int edad = (int)temp;
```

2.11.2.2 Estándares metodológicos

- **CERT/C-ERR34**: Detect errors when converting a string to a number.

2.12 Concurrencia

2.12.1 [c.sync.1] - No destruir un mutex mientras esté bloqueado

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

Los `mutex` proporcionan sincronización entre hilos mediante operaciones de bloqueo y desbloqueo. Destruir un `mutex` mientras permanece bloqueado por algún hilo puede provocar **bloqueos permanentes, corrupción de memoria o fallos del sistema**. Esta situación surge frecuentemente cuando no se garantiza el desbloqueo antes de la destrucción del objeto `mutex`.

Codificación insegura

Uso de un `mutex` para proteger una sección crítica entre hilos, mostrando qué ocurre cuando se intenta liberar el recurso de sincronización sin haberlo desbloqueado

previamente. El objetivo es entender el impacto en estabilidad y comportamiento indefinido cuando no se respeta el ciclo correcto de bloqueo y desbloqueo.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;

int main(void) {
  //-- Inicialización del mutex
  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("Error al inicializar el mutex");
    return EXIT_FAILURE;
  }
  //-- Bloqueo del mutex antes de entrar en la sección crítica
  if (pthread_mutex_lock(&mutex) != 0) {
    perror("Error al bloquear el mutex");
    return EXIT_FAILURE;
  }
  //-- Sección crítica protegida
  printf("Sección crítica en ejecución\n");

  //-- Destrucción del mutex sin haberlo desbloqueado (comportamiento indefinido)
  if (pthread_mutex_destroy(&mutex) != 0) {
    perror("Error al destruir el mutex");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Uso correcto de un `mutex` respetando su ciclo de vida completo: inicialización, bloqueo, desbloqueo y destrucción. Se muestra cómo liberar explícitamente el bloqueo antes de destruir el recurso para evitar comportamiento indefinido y garantizar un cierre limpio del programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;

int main(void) {
  //-- Inicialización del mutex
  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("Error al inicializar el mutex");
```

```
    return EXIT_FAILURE;
  }
  //-- Bloqueo del mutex antes de entrar en la sección crítica
  if (pthread_mutex_lock(&mutex) != 0) {
    perror("Error al bloquear el mutex");
    pthread_mutex_destroy(&mutex);
    return EXIT_FAILURE;
  }
  //-- Sección crítica protegida
  printf("Sección crítica en ejecución\n");

  //-- Desbloqueo del mutex antes de destruirlo
  if (pthread_mutex_unlock(&mutex) != 0) {
    perror("Error al desbloquear el mutex");
    pthread_mutex_destroy(&mutex);
    return EXIT_FAILURE;
  }
  //-- Destrucción segura del mutex una vez liberado
  if (pthread_mutex_destroy(&mutex) != 0) {
    perror("Error al destruir el mutex");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

2.12.1.1 Descripción

Los `mutex` proporcionan sincronización entre hilos mediante mecanismos de bloqueo y desbloqueo. La destrucción de un `mutex` mientras permanece bloqueado genera **comportamiento indefinido** en el programa.

Cuando un hilo destruye un `mutex` bloqueado, el sistema no puede garantizar la liberación correcta de los recursos asociados. Esta situación compromete la integridad del mecanismo de sincronización y puede provocar **corrupción de memoria**, bloqueos permanentes de hilos o terminación inesperada del programa.

El problema surge frecuentemente en escenarios donde múltiples hilos acceden al mismo `mutex` y uno de ellos intenta destruirlo sin verificar su estado. También ocurre durante la limpieza de recursos al finalizar el programa, cuando se destruyen estructuras que contienen `mutex` sin considerar su estado de bloqueo.

La función `pthread_mutex_destroy()` no verifica automáticamente si el `mutex` está bloqueado antes de proceder con la destrucción. Esta responsabilidad recae completamente en el programador, quien debe implementar los controles necesarios.

La mitigación requiere establecer protocolos claros para la gestión del ciclo de vida de los `mutex`. Esto incluye verificar que todos los hilos hayan liberado el `mutex` antes de su

destrucción y coordinar adecuadamente las operaciones de bloqueo, desbloqueo y destrucción entre los diferentes hilos del programa.

El diseño de la aplicación debe considerar el **orden de destrucción** de los recursos compartidos y establecer mecanismos que garanticen la sincronización correcta durante las fases de inicialización y finalización del programa.

Riesgos asociados

- Condiciones de carrera que pueden llevar a corrupción de datos.
- Fugas de recursos debido a hilos bloqueados indefinidamente.
- Bloqueos y deadlocks por hilos esperando un `mutex` destruido.
- Comportamiento indefinido del programa ante accesos concurrentes.
- Caídas inesperadas de la aplicación (crashes) por acceso a memoria liberada o recursos inexistentes.
- Imposibilidad de realizar depuración efectiva por errores no deterministas.
- Vulnerabilidades explotables que podrían comprometer la estabilidad o seguridad de la aplicación.

Buenas prácticas

- Verificar que el `mutex` esté desbloqueado antes de destruirlo, asegurando que ningún hilo lo esté utilizando.

```
//-- Destruir el mutex sin garantizar que esté desbloqueado
pthread_mutex_lock(&mtx);
pthread_mutex_destroy(&mtx);
```

```
//-- Desbloquear el mutex antes de proceder a su destrucción
pthread_mutex_lock(&mtx);
pthread_mutex_unlock(&mtx);
pthread_mutex_destroy(&mtx);
```

- Coordinar la finalización de hilos antes de destruir el `mutex` para evitar que algún hilo intente bloquearlo posteriormente.

```
//-- Destruir el mutex mientras otros hilos pueden seguir ejecutándose
pthread_create(&th, NULL, rutina, NULL);
pthread_mutex_destroy(&mtx);
```

```
//-- Esperar a que los hilos finalicen antes de destruir el mutex
pthread_create(&th, NULL, rutina, NULL);
pthread_join(th, NULL);
```

```
pthread_mutex_destroy(&mtx);
```

- Implementar un mecanismo de control de estado para evitar múltiples destrucciones o destrucción prematura del `mutex`.

```
//-- Destruir el mutex sin verificar su estado de inicialización
if(error_condicion) {
  pthread_mutex_destroy(&mtx);
}
```

```
//-- Verificar que el mutex esté inicializado y no esté en uso antes de destruirlo
if(mtx_inicializado) {
  pthread_mutex_lock(&mtx);
  pthread_mutex_unlock(&mtx);
  pthread_mutex_destroy(&mtx);
  mtx_inicializado = 0;
}
```

2.12.1.2 Estándares metodológicos

- **CERT/C-CON31**: Do not destroy a mutex while it is locked.

2.12.2 [c.sync.2] - Prevenir condiciones de carrera de datos en código multihilo

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

Los programas multihilo que acceden concurrentemente a datos compartidos sin sincronización adecuada pueden experimentar **condiciones de carrera de datos**. Estas condiciones provocan que el resultado de las operaciones dependa del orden impredecible de ejecución de los hilos, generando comportamientos inconsistentes y errores difíciles de reproducir y depurar.

Codificación insegura

Uso de hilos que incrementan un contador global compartido para mostrar cómo el acceso concurrente sin sincronización puede provocar resultados inconsistentes. Se pretende observar el efecto de múltiples hilos modificando el mismo recurso sin ningún mecanismo de protección.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
```

```
//-- contador global compartido por todos los hilos
int contador = 0;

//-- función que incrementa el contador sin sincronización
void* incrementar(void* arg) {
  //-- bucle que incrementa el contador muchas veces
  for (int i = 0; i < 100000; i++) {
    contador++;
  }
  return NULL;
}
int main() {
  pthread_t h1, h2;

  //-- creación de dos hilos que acceden al mismo recurso
  if (pthread_create(&h1, NULL, incrementar, NULL) != 0) {
    perror("Error creando h1");
    exit(EXIT_FAILURE);
  }
  if (pthread_create(&h2, NULL, incrementar, NULL) != 0) {
    perror("Error creando h2");
    exit(EXIT_FAILURE);
  }
  //-- espera a que ambos hilos finalicen
  pthread_join(h1, NULL);
  pthread_join(h2, NULL);

  //-- el resultado puede ser inconsistente por carrera de datos
  printf("Valor final del contador: %d\n", contador);
  return 0;
}
```

Codificación recomendada

Uso de hilos que incrementan un contador global compartido aplicando un mecanismo de exclusión mutua para garantizar que solo un hilo accede al recurso crítico a la vez. Se busca asegurar consistencia en los datos y eliminar condiciones de carrera.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

//-- contador global compartido por todos los hilos
int contador = 0;

//-- mutex para proteger la sección crítica
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
//-- función que incrementa el contador con sincronización
void* incrementar(void* arg) {
  //-- bucle que incrementa el contador muchas veces
  for (int i = 0; i < 100000; i++) {
    //-- inicio de sección crítica protegida
    pthread_mutex_lock(&mutex);
    contador++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
int main() {
  pthread_t h1, h2;

  //-- creación de dos hilos que acceden al mismo recurso protegido
  if (pthread_create(&h1, NULL, incrementar, NULL) != 0) {
    perror("Error creando h1");
    exit(EXIT_FAILURE);
  }
  if (pthread_create(&h2, NULL, incrementar, NULL) != 0) {
    perror("Error creando h2");
    exit(EXIT_FAILURE);
  }
  //-- espera a que ambos hilos finalicen
  pthread_join(h1, NULL);
  pthread_join(h2, NULL);

  //-- el resultado será consistente gracias a la exclusión mutua
  printf("Valor final del contador: %d\n", contador);

  //-- destrucción del mutex antes de finalizar
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

2.12.2.1 Descripción

Las condiciones de carrera de datos ocurren cuando múltiples hilos acceden simultáneamente a la misma ubicación de memoria y al menos uno de ellos realiza una operación de escritura, sin que exista sincronización adecuada entre los accesos. Esta situación genera comportamiento indefinido según el estándar del lenguaje.

El problema surge porque el orden de ejecución de las operaciones entre diferentes hilos no está garantizado. Cuando dos hilos intentan modificar o leer una variable compartida sin coordinación, el resultado final depende del momento exacto en que cada hilo ejecuta sus instrucciones, creando resultados impredecibles.

Los riesgos asociados incluyen **corrupción de datos**, donde los valores almacenados pueden quedar en estados inconsistentes o parcialmente actualizados. También pueden producirse **fallos de programa** debido al comportamiento indefinido, y **vulnerabilidades de seguridad** cuando los datos corruptos afectan a validaciones o controles de acceso.

La detección de condiones de carrera de datos presenta dificultades significativas durante las pruebas, ya que pueden manifestarse de forma intermitente y depender de factores como la carga del sistema o el planificador de hilos del sistema operativo.

La mitigación requiere implementar **mecanismos de sincronización** apropiados para coordinar el acceso a recursos compartidos. Las estrategias incluyen el uso de primitivas de sincronización como `mutex`, semáforos o variables de condición, así como técnicas de programación que minimicen el estado compartido entre hilos.

También resulta efectivo aplicar **herramientas de análisis estático y dinámico** especializadas en la detección de condiciones de carrera, junto con revisiones de código enfocadas en identificar patrones de acceso concurrente no protegido.

Riesgos asociados

- Condiciones de carrera que provocan resultados impredecibles y errores de lógica en la aplicación.
- Pérdida o corrupción de datos compartidos entre hilos.
- Fallos de seguridad por explotación de estados inconsistentes de la memoria.
- Introducción de vulnerabilidades como escalada de privilegios o ejecución de código malicioso.
- Bloqueos o deadlocks involuntarios, afectando la disponibilidad del sistema.
- Dificultad en la detección y depuración de errores, incrementando el costo de mantenimiento.
- Comportamiento no determinista que genera fallas intermitentes difíciles de reproducir.
- Condiciones propicias para ataques de tipo Time-of-Check Time-of-Use (TOCTOU).
- Desviación no intencionada del flujo de control del programa.

Buenas prácticas

- Proteger las secciones críticas con mecanismos de sincronización como `mutex` para evitar accesos concurrentes a recursos compartidos.

```
//-- Acceso a variable compartida sin sincronización
int contador = 0;
void incrementar() {
```

```
  contador++;
}
```

```
//-- Protección de la sección crítica con mutex
int contador = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void incrementar() {
  pthread_mutex_lock(&lock);
  contador++;
  pthread_mutex_unlock(&lock);
}
```

- Utilizar variables atómicas para operaciones simples compartidas entre hilos.

```
//-- Incremento no atómico en entorno multihilo
int bandera = 0;
void activar() {
  bandera = 1;
}
```

```
//-- Uso de tipo atómico para operación segura entre hilos
_Atomic int bandera = 0;
void activar() {
  atomic_store(&bandera, 1);
}
```

- Evitar compartir datos mutables entre hilos cuando no sea necesario, favoreciendo almacenamiento local por hilo.

```
//-- Uso de variable global compartida entre hilos
int resultado;
void calcular() {
  resultado = 42;
}
```

```
//-- Uso de almacenamiento local para cada hilo
void calcular() {
  int resultado_local = 42;
  /* uso interno del resultado */
}
```

- Establecer un orden consistente de adquisición de bloqueos para prevenir condiciones de carrera y estados inconsistentes.

```
//-- Bloqueo de recursos en orden inconsistente
```

```
pthread_mutex_t lockA, lockB;
void tarea1() {
  pthread_mutex_lock(&lockA);
  pthread_mutex_lock(&lockB);
}
void tarea2() {
  pthread_mutex_lock(&lockB);
  pthread_mutex_lock(&lockA);
}
```

```
//-- Bloqueo de recursos siguiendo un orden único y definido
pthread_mutex_t lockA, lockB;
void tarea1() {
  pthread_mutex_lock(&lockA);
  pthread_mutex_lock(&lockB);
}
void tarea2() {
  pthread_mutex_lock(&lockA);
  pthread_mutex_lock(&lockB);
}
```

2.12.2.2 Estándares metodológicos

- **CERT/C-CON32**: Prevent data races when accessing bit-fields from multiple threads.
- **CERT/C-CON43**: Do not allow data races in multithreaded code.

2.12.3 [c.sync.3] - Evitar interbloqueos bloqueando mutex en un orden predefinido

ID	c.sync.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Los **interbloqueos** representan uno de los problemas más críticos en programación concurrente, ocurriendo cuando dos o más hilos quedan bloqueados indefinidamente esperando recursos que otros hilos mantienen. Esta situación surge frecuentemente al adquirir múltiples `mutex` sin seguir un orden consistente, creando dependencias circulares que paralizan la ejecución del programa y comprometen su estabilidad.

Codificación insegura

Se crean dos hilos que necesitan acceder a dos recursos compartidos protegidos por `mutex`. El objetivo es mostrar cómo el orden en el que se adquieren los bloqueos influye

directamente en la posibilidad de que el programa quede bloqueado permanentemente cuando varios hilos compiten por los mismos recursos.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread_func1(void* arg) {
  //-- Bloquea primero mutex1
  pthread_mutex_lock(&mutex1);
  sleep(1);

  //-- Intenta bloquear mutex2
  pthread_mutex_lock(&mutex2);

  printf("Hilo 1 obtuvo ambos mutex\n");

  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
void* thread_func2(void* arg) {
  //-- Bloquea primero mutex2
  pthread_mutex_lock(&mutex2);
  sleep(1);

  //-- Intenta bloquear mutex1
  pthread_mutex_lock(&mutex1);

  printf("Hilo 2 obtuvo ambos mutex\n");

  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
  return NULL;
}
int main() {
  pthread_t t1, t2;

  pthread_create(&t1, NULL, thread_func1, NULL);
  pthread_create(&t2, NULL, thread_func2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  pthread_mutex_destroy(&mutex1);
  pthread_mutex_destroy(&mutex2);
```

```
  return 0;
}
```

Codificación recomendada

Se crean dos hilos que necesitan acceder a los mismos recursos compartidos. Se define un orden global y predecible para adquirir los `mutex`, garantizando que todos los hilos sigan la misma secuencia de bloqueo y evitando así condiciones de interbloqueo.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread_func1(void* arg) {
  //-- Bloquea mutex1 siguiendo el orden global definido
  pthread_mutex_lock(&mutex1);
  sleep(1);

  //-- Luego bloquea mutex2 respetando el mismo orden
  pthread_mutex_lock(&mutex2);

  printf("Hilo 1 obtuvo ambos mutex\n");

  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
void* thread_func2(void* arg) {
  //-- Bloquea mutex1 siguiendo el mismo orden global
  pthread_mutex_lock(&mutex1);
  sleep(1);

  //-- Luego bloquea mutex2 respetando el mismo orden
  pthread_mutex_lock(&mutex2);

  printf("Hilo 2 obtuvo ambos mutex\n");

  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
int main() {
  pthread_t t1, t2;
```

```
    pthread_create(&t1, NULL, thread_func1, NULL);
    pthread_create(&t2, NULL, thread_func2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

2.12.3.1 Descripción

El problema fundamental ocurre cuando el hilo A bloquea el `mutex` 1 y espera el `mutex` 2, mientras que simultáneamente el hilo B bloquea el `mutex` 2 y espera el `mutex` 1. Ninguno de los dos hilos puede continuar su ejecución, creando una dependencia circular que paraliza el programa.

Esta regla establece que todos los hilos deben adquirir múltiples `mutex` siguiendo un **orden consistente y predefinido** a lo largo de toda la aplicación. La ordenación puede basarse en criterios como direcciones de memoria, identificadores numéricos o cualquier esquema que garantice una secuencia determinística.

Los riesgos asociados al incumplimiento de esta regla incluyen la paralización completa del programa, degradación del rendimiento por timeouts y reintentos, y comportamientos impredecibles que son especialmente difíciles de reproducir y depurar en entornos de producción.

La mitigación efectiva requiere establecer una política de ordenación clara desde el diseño del sistema, documentar explícitamente el orden de adquisición de recursos e implementar mecanismos de verificación durante el desarrollo. Algunas implementaciones utilizan técnicas adicionales como timeouts en las operaciones de bloqueo o algoritmos de detección de interbloqueos para proporcionar capas extra de protección.

Esta práctica resulta especialmente crítica en sistemas embebidos y aplicaciones de tiempo real, donde los interbloqueos pueden tener consecuencias graves más allá de la simple degradación del rendimiento.

Riesgos asociados

- Interbloqueo entre hilos (deadlock), donde múltiples threads quedan bloqueados esperando recursos que nunca serán liberados.
- Bloqueo indefinido de la aplicación o pérdida de disponibilidad del servicio/software.

- Dificultad para detectar y depurar el origen de los bloqueos durante el desarrollo o en producción.
- Degradación del rendimiento debido a hilos en espera indefinida, lo cual puede afectar a otras partes del sistema.
- Mayor consumo de recursos por hilos bloqueados, llegando a agotar memoria o CPU.
- Pérdida de datos o transacciones incompletas si los bloqueos ocurren durante operaciones críticas.
- Riesgo de race conditions si se intentan soluciones ad-hoc o no estructuradas para evitar los deadlocks.
- Vulnerabilidades de denegación de servicio (DoS), ya que un atacante podría provocar un deadlock intencionadamente.

Buenas prácticas

- Definir y documentar un orden global de adquisición de `mutex` y respetarlo en todos los hilos.

```
//-- Bloqueo de mutex en orden variable según el flujo
pthread_mutex_lock(&mutexB);
pthread_mutex_lock(&mutexA);
/* sección crítica */
pthread_mutex_unlock(&mutexA);
pthread_mutex_unlock(&mutexB);
```

```
//-- Bloqueo siguiendo un orden global predefinido (A luego B)
pthread_mutex_lock(&mutexA);
pthread_mutex_lock(&mutexB);
/* sección crítica */
pthread_mutex_unlock(&mutexB);
pthread_mutex_unlock(&mutexA);
```

- Utilizar identificadores jerárquicos para determinar dinámicamente el orden de bloqueo.

```
//-- Cada hilo bloquea según parámetros externos sin criterio común
pthread_mutex_lock(first);
pthread_mutex_lock(second);
/* sección crítica */
pthread_mutex_unlock(second);
pthread_mutex_unlock(first);
```

```
//-- Se ordenan los mutex por dirección de memoria antes de bloquear
```

```
if (first < second) {
  pthread_mutex_lock(first);
  pthread_mutex_lock(second);
} else {
  pthread_mutex_lock(second);
  pthread_mutex_lock(first);
}
/* sección crítica */
pthread_mutex_unlock(first);
pthread_mutex_unlock(second);
```

- Evitar funciones que bloqueen `mutex` adicionales sin respetar el orden global establecido.

```
//-- Función interna bloquea mutex sin considerar el orden ya adquirido
pthread_mutex_lock(&mutexA);
procesar(); /* dentro bloquea mutexB */
pthread_mutex_unlock(&mutexA);
```

```
//-- Función recibe mutex ya bloqueados respetando el orden global
pthread_mutex_lock(&mutexA);
pthread_mutex_lock(&mutexB);
procesar_sin_bloqueo_adicional();
pthread_mutex_unlock(&mutexB);
pthread_mutex_unlock(&mutexA);
```

- Implementar mecanismos de intento de bloqueo con liberación controlada si no se puede mantener el orden.

```
//-- Bloqueo directo sin control puede causar espera circular
pthread_mutex_lock(&mutexA);
pthread_mutex_lock(&mutexB);
/* sección crítica */
```

```
//-- Uso de trylock y liberación si no se obtiene el segundo mutex
pthread_mutex_lock(&mutexA);
if (pthread_mutex_trylock(&mutexB) != 0) {
  pthread_mutex_unlock(&mutexA);
  /* reintentar siguiendo política definida */
} else {
  /* sección crítica */
  pthread_mutex_unlock(&mutexB);
  pthread_mutex_unlock(&mutexA);
}
```

2.12.3.2 Estándares metodológicos

- **CERT/C-CON35**: Avoid deadlock by locking in a predefined order.

2.12.4 [c.sync.4] - No volver a unir o desvincular un hilo previamente unido o desvinculado

ID	**c.sync.4**	**Uso**	**Obligatorio**
CWSS Score	**74**	**Impacto técnico**	**Alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los hilos de ejecución requieren una gestión precisa de su ciclo de vida para evitar comportamientos indefinidos y fugas de recursos. Intentar realizar operaciones de `join` o `detach` múltiples sobre el mismo hilo, o combinar ambas operaciones, genera **condiciones de error** que comprometen la estabilidad del programa y pueden provocar fallos en tiempo de ejecución.

Codificación insegura

Gestión de hilos utilizando la biblioteca POSIX `pthread`, mostrando el impacto de intentar unir (`join`) un hilo más de una vez. Se pretende entender cómo una mala gestión del ciclo de vida del hilo puede provocar errores en tiempo de ejecución y comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* tarea(void* arg) {
  //-- Simula trabajo del hilo
  printf("Hilo en ejecucion\n");
  return NULL;
}
int main() {
  pthread_t hilo;
  int resultado;

  //-- Creacion del hilo
  if (pthread_create(&hilo, NULL, tarea, NULL) != 0) {
    perror("Error al crear el hilo");
    return EXIT_FAILURE;
  }
  //-- Primera union del hilo
  resultado = pthread_join(hilo, NULL);
  if (resultado != 0) {
    perror("Error en pthread_join");
    return EXIT_FAILURE;
```

```
  }
  //-- Segunda union del mismo hilo (error)
  resultado = pthread_join(hilo, NULL);
  if (resultado != 0) {
    perror("Segundo pthread_join fallido");
  }
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Gestión segura del ciclo de vida de un hilo, asegurando que solo se realice la operación de unión una vez y controlando el estado del hilo mediante una variable de seguimiento. Se busca evitar errores por uniones múltiples y mantener un flujo de ejecución predecible.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>

void* tarea(void* arg) {
  //-- Simula trabajo del hilo
  printf("Hilo en ejecucion\n");
  return NULL;
}
int main() {
  pthread_t hilo;
  int resultado;
  bool hilo_unido = false;

  //-- Creacion del hilo
  if (pthread_create(&hilo, NULL, tarea, NULL) != 0) {
    perror("Error al crear el hilo");
    return EXIT_FAILURE;
  }
  //-- Union controlada del hilo
  if (!hilo_unido) {
    resultado = pthread_join(hilo, NULL);
    if (resultado != 0) {
      perror("Error en pthread_join");
      return EXIT_FAILURE;
    }
    hilo_unido = true;
  }
  //-- No se intenta unir nuevamente el hilo

  return EXIT_SUCCESS;
```

```
}
```

2.12.4.1 Descripción

Esta regla establece que cada hilo debe ser unido o separado exactamente una vez durante su ciclo de vida. Un hilo que ya ha sido objeto de una operación `pthread_join()` o `pthread_detach()` no debe volver a ser procesado con ninguna de estas funciones.

El sistema de gestión de hilos mantiene un estado interno para cada hilo que determina si puede ser unido o separado. Una vez que se ejecuta `pthread_join()` sobre un hilo, este queda marcado como procesado y sus recursos son liberados. De manera similar, `pthread_detach()` marca el hilo para liberación automática de recursos al finalizar su ejecución.

Intentar realizar operaciones duplicadas sobre el mismo hilo genera **comportamiento indefinido** según el estándar POSIX. El sistema puede responder de manera impredecible, desde fallos silenciosos hasta terminación abrupta del programa. Los recursos del hilo pueden quedar en un estado inconsistente, afectando la estabilidad general de la aplicación.

Los riesgos asociados incluyen corrupción de estructuras de datos internas del sistema de hilos, liberación múltiple de memoria, y condiciones de carrera en el gestor de hilos. Estos problemas pueden manifestarse de forma intermitente, complicando la detección y depuración.

La mitigación efectiva requiere un **diseño claro de responsabilidades** para la gestión del ciclo de vida de cada hilo. Se debe establecer qué componente del código es responsable de unir o separar cada hilo específico. El uso de variables de estado o estructuras de control puede ayudar a rastrear el estado de procesamiento de cada hilo y prevenir operaciones duplicadas.

La implementación de verificaciones de estado antes de ejecutar operaciones de unión o separación proporciona una capa adicional de protección contra este tipo de errores.

Riesgos asociados

- Condiciones de carrera y comportamiento indefinido en la ejecución del programa.
- Acceso concurrente a recursos compartidos sin sincronización adecuada.
- Fugas de memoria o recursos debido a hilos huérfanos o mal gestionados.
- Pérdida de información de retorno o estado de los hilos.
- Aparición de bloqueos o deadlocks por mal manejo del ciclo de vida del hilo.
- Inestabilidad o fallos inesperados en el sistema (crash).

- Violación de políticas de seguridad debido a ejecución no controlada de código.
- Dificultad en la depuración y mantenimiento del software por estados inconsistentes.

Buenas prácticas

- Verificar y registrar el estado del hilo para evitar llamar a `pthread_join()` o `pthread_detach()` más de una vez sobre el mismo identificador.

```
//-- Unión duplicada del mismo hilo sin control de estado
pthread_t tid;
pthread_create(&tid, NULL, worker, NULL);
pthread_join(tid, NULL);
pthread_join(tid, NULL);
```

```
//-- Control explícito del estado del hilo para evitar doble unión
pthread_t tid;
int joined = 0;
pthread_create(&tid, NULL, worker, NULL);
if (!joined) {
  pthread_join(tid, NULL);
  joined = 1;
}
```

- No separar (`pthread_detach()`) un hilo que ya fue unido, y definir claramente la estrategia de gestión (unir o separar) desde su creación.

```
//-- Separación de un hilo después de haber sido unido
pthread_t tid;
pthread_create(&tid, NULL, worker, NULL);
pthread_join(tid, NULL);
pthread_detach(tid);
```

```
//-- Definición única de estrategia: hilo separado desde el inicio
pthread_t tid;
pthread_create(&tid, NULL, worker, NULL);
pthread_detach(tid);
```

2.12.4.2 Estándares metodológicos

- **CERT/C-CON39**: Do not join or detach a thread that was previously joined or detached.

2.13 POSIX

2.13.1 [c.posix.1] - No llamar a putenv con punteros a variables automáticas

ID	c.posix.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La función `putenv` almacena directamente el puntero proporcionado en el entorno del programa sin realizar una copia de los datos. Cuando se pasan **variables automáticas** como argumentos, estas pueden ser destruidas al salir del ámbito donde fueron declaradas, dejando punteros inválidos en el entorno y variables de entorno corruptas.

Codificación insegura

Se muestra cómo establecer una variable de entorno utilizando `putenv` con un buffer local definido dentro de una función. El objetivo es observar qué ocurre cuando se pasa a `putenv` un puntero a memoria automática que deja de ser válida al salir del bloque, generando comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void configurar_variable() {
  //-- se define un buffer automático en la pila
  char variable[50];

  //-- se construye la cadena con el formato requerido por putenv
  snprintf(variable, sizeof(variable), "MI_VARIABLE=%s", "valor_inseguro");

  //-- se pasa a putenv un puntero a memoria automática
  if (putenv(variable) != 0) {
    perror("Error en putenv");
  }
}
int main() {
  //-- se llama a la función que configura la variable
  configurar_variable();

  //-- se intenta leer la variable de entorno
  char *valor = getenv("MI_VARIABLE");
  if (valor != NULL) {
    printf("MI_VARIABLE=%s\n", valor);
  } else {
    printf("MI_VARIABLE no definida\n");
  }
```

```
  return 0;
}
```

Codificación recomendada

Se presenta la forma segura de establecer una variable de entorno asegurando que la memoria asociada permanezca válida después de la llamada. Se utiliza `setenv`, que gestiona internamente la memoria, evitando depender de variables automáticas y eliminando el riesgo de comportamiento indefinido.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  //-- se establece la variable de entorno de forma segura
  if (setenv("MI_VARIABLE", "valor_seguro", 1) != 0) {
    perror("Error en setenv");
    return 1;
  }
  //-- se obtiene el valor de la variable de entorno
  char *valor = getenv("MI_VARIABLE");
  if (valor != NULL) {
    printf("MI_VARIABLE=%s\n", valor);
  } else {
    printf("MI_VARIABLE no definida\n");
  }
  return 0;
}
```

2.13.1.1 Descripción

La función `putenv` modifica el entorno del proceso añadiendo o cambiando variables de entorno mediante un puntero a una cadena de caracteres. Esta función no realiza una copia de la cadena proporcionada, sino que almacena directamente el puntero en la tabla de entorno del sistema.

Cuando se pasa un puntero a una **variable automática** (variable local almacenada en la pila), se crea una situación de riesgo crítico. Las variables automáticas tienen un ciclo de vida limitado al ámbito de la función donde se declaran y se destruyen automáticamente al finalizar dicha función.

El problema surge porque `putenv` mantiene la referencia al puntero incluso después de que la variable automática haya salido de su ámbito. Esto resulta en un **puntero colgante** en la tabla de entorno, que apunta a una región de memoria que ya no contiene los datos originales y puede haber sido reutilizada para otros propósitos.

Los riesgos asociados incluyen comportamiento indefinido del programa, corrupción de datos, fallos de segmentación y potenciales vulnerabilidades de seguridad. El acceso posterior a estas variables de entorno puede producir resultados impredecibles o comprometer la estabilidad del sistema.

La mitigación de este problema requiere garantizar que los datos pasados a `putenv` mantengan su validez durante toda la ejecución del programa. Esto se logra utilizando memoria con duración estática, memoria asignada dinámicamente, o variables globales en lugar de variables automáticas locales.

Riesgos asociados

- Condición de carrera que puede llevar a resultados impredecibles debido a la reutilización de memoria local.
- Corrupción de la tabla de entorno al quedar punteros colgados cuando sale del alcance la variable automática.
- Posible utilización de datos no inicializados o erróneos por otras partes del programa que consulten las variables de entorno.
- Exposición a vulnerabilidades de seguridad, como ejecución de código arbitrario, si un atacante logra controlar el contenido liberado o reutilizado.
- Errores de difícil depuración y comportamiento indefinido que pueden provocar fallos intermitentes del sistema.
- Filtración accidental de información sensible si la memoria liberada es reasignada y leída posteriormente como parte del entorno.

Buenas prácticas

- No pasar a `putenv` punteros a variables automáticas (stack) que puedan quedar inválidas tras salir del ámbito.

```
//-- Uso de variable automática cuyo ciclo de vida finaliza al salir de la función
void configurar() {
  char buffer[64];
  snprintf(buffer, sizeof(buffer), "PATH=%s", "/tmp");
  putenv(buffer);
}
```

```
//-- Uso de memoria dinámica que permanece válida después de la llamada
void configurar() {
  char *buffer = malloc(64);
  if (buffer != NULL) {
    snprintf(buffer, 64, "PATH=%s", "/tmp");
    putenv(buffer);
  }
```

```
}
```

- Preferir `setenv` en lugar de `putenv` para evitar dependencia del almacenamiento externo.

```
//-- Dependencia del almacenamiento externo proporcionado por el llamador
char *buffer = malloc(64);
snprintf(buffer, 64, "HOME=%s", "/home/user");
putenv(buffer);
```

```
//-- Copia interna del valor gestionada por la implementación
setenv("HOME", "/home/user", 1);
```

- No modificar ni liberar la memoria pasada a `putenv` después de establecer la variable de entorno.

```
//-- Liberación de memoria aún referenciada por el entorno
char *buffer = malloc(64);
snprintf(buffer, 64, "LANG=%s", "C");
putenv(buffer);
free(buffer);
```

```
//-- Mantener la memoria válida durante toda la vida del proceso
static char buffer[64];
snprintf(buffer, sizeof(buffer), "LANG=%s", "C");
putenv(buffer);
```

- Validar la longitud y formato de la cadena antes de invocar `putenv` para evitar desbordamientos o variables mal formadas.

```
//-- Construcción sin control de límites
char *buffer = malloc(16);
sprintf(buffer, "USER=%s", nombre_usuario);
putenv(buffer);
```

```
//-- Construcción con verificación de tamaño y formato esperado
char *buffer = malloc(64);
if (buffer != NULL) {
  snprintf(buffer, 64, "USER=%s", nombre_usuario);
  putenv(buffer);
}
```

2.13.1.2 Estándares metodológicos

- **CERT/C-POS34**: Do not call putenv() with a pointer to an automatic variable as

the argument.

2.13.2 [c.posix.2] - Evitar condiciones de carrera al comprobar enlaces simbólicos

ID	c.posix.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los enlaces simbólicos pueden modificarse entre el momento de su verificación y su uso posterior, creando una ventana temporal donde atacantes pueden redirigir operaciones hacia archivos no autorizados. Esta **condición de carrera** permite que programas con privilegios elevados accedan inadvertidamente a recursos sensibles del sistema, comprometiendo la seguridad mediante ataques de tipo **TOCTOU** (Time-of-Check-Time-of-Use).

Codificación insegura

Acceso a un archivo proporcionado por el usuario validando primero que no sea un enlace simbólico y luego escribiendo en él. Se busca mostrar cómo una validación separada del uso puede permitir que un atacante cambie el archivo entre ambas operaciones, generando una condición de carrera (TOCTOU).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

//-- Comprueba si el archivo es un enlace simbólico usando lstat
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Uso: %s <archivo>\n", argv[0]);
    return EXIT_FAILURE;
  }
  struct stat st;

  //-- Verifica el tipo de archivo antes de abrirlo
  if (lstat(argv[1], &st) == -1) {
    perror("lstat");
    return EXIT_FAILURE;
  }
  if (S_ISLNK(st.st_mode)) {
    fprintf(stderr, "No se permiten enlaces simbólicos\n");
    return EXIT_FAILURE;
  }
```

```
  //-- Abre el archivo después de la comprobación
  int fd = open(argv[1], O_WRONLY | O_CREAT, 0644);
  if (fd == -1) {
    perror("open");
    return EXIT_FAILURE;
  }
  //-- Escribe datos en el archivo
  const char *data = "Datos sensibles\n";
  if (write(fd, data, strlen(data)) == -1) {
    perror("write");
    close(fd);
    return EXIT_FAILURE;
  }
  close(fd);
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Acceso seguro a un archivo evitando condiciones de carrera al impedir que el sistema siga enlaces simbólicos en la propia llamada a apertura. Se integra la validación y el uso en una única operación atómica y se verifica posteriormente el descriptor abierto.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

//-- Abre el archivo evitando seguir enlaces simbólicos
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Uso: %s <archivo>\n", argv[0]);
    return EXIT_FAILURE;
  }
  //-- O_NOFOLLOW evita que open siga enlaces simbólicos
  int fd = open(argv[1], O_WRONLY | O_CREAT | O_NOFOLLOW, 0644);
  if (fd == -1) {
    perror("open");
    return EXIT_FAILURE;
  }
  struct stat st;

  //-- Verifica el descriptor ya abierto
  if (fstat(fd, &st) == -1) {
    perror("fstat");
    close(fd);
```

```
    return EXIT_FAILURE;
  }
  if (!S_ISREG(st.st_mode)) {
    fprintf(stderr, "El archivo no es regular\n");
    close(fd);
    return EXIT_FAILURE;
  }
  //-- Escribe datos de forma segura en el archivo validado
  const char *data = "Datos sensibles\n";
  if (write(fd, data, strlen(data)) == -1) {
    perror("write");
    close(fd);
    return EXIT_FAILURE;
  }
  close(fd);
  return EXIT_SUCCESS;
}
```

2.13.2.1 Descripción

Las condiciones de carrera en la verificación de enlaces simbólicos ocurren cuando existe un intervalo temporal entre la comprobación de las propiedades de un archivo y su posterior uso. Durante este intervalo, un atacante puede modificar el enlace simbólico para que apunte a un archivo diferente, provocando que el programa opere sobre un recurso no intencionado.

Este problema surge principalmente en operaciones que involucran funciones como `lstat()`, `stat()` o `access()` seguidas de llamadas a `open()`, `fopen()` u otras funciones de manipulación de archivos. El sistema operativo no garantiza que el archivo permanezca inalterado entre estas operaciones, creando una ventana de vulnerabilidad.

Los riesgos asociados incluyen **escalada de privilegios**, **acceso no autorizado a archivos sensibles** y **corrupción de datos**. Un atacante puede explotar estas condiciones para redirigir operaciones de escritura hacia archivos críticos del sistema o para acceder a información confidencial mediante la manipulación de enlaces simbólicos en directorios con permisos de escritura compartidos.

La mitigación efectiva requiere el uso de **operaciones atómicas** que combinen la verificación y el acceso en una sola llamada al sistema. Las funciones de la familia `openat()` y el uso de descriptores de archivo proporcionan mecanismos para evitar estas condiciones de carrera. Adicionalmente, la configuración adecuada de permisos de directorio y la validación de rutas pueden reducir la superficie de ataque.

La implementación de controles de acceso basados en **descriptores de archivo** en lugar de nombres de archivo elimina la dependencia de la resolución de rutas durante operaciones críticas, proporcionando un enfoque más seguro para el manejo de recursos del sistema de archivos.

Riesgos asociados

- Escalada de privilegios por manipulación de enlaces simbólicos.
- Divulgación de información sensible por acceso a archivos incorrectos.
- Modificación o corrupción de datos por redireccionamiento de archivos.
- Bypass o evasión de controles de acceso.
- Ejecución de código malicioso a través de archivos substituidos.
- Intercambio de archivos durante operaciones críticas (TOCTOU).
- Inyección de contenido inesperado por parte de usuarios locales o atacantes.

Buenas prácticas

- Evitar la verificación separada del uso (TOCTOU) utilizando descriptores de archivo en lugar de rutas al acceder a enlaces simbólicos.

```
//-- Verificación previa con lstat() y apertura posterior por ruta
struct stat st;
if (lstat("archivo.txt", &st) == 0) {
  if (!S_ISLNK(st.st_mode)) {
    int fd = open("archivo.txt", O_RDONLY);
  }
}
```

```
//-- Apertura directa con O_NOFOLLOW y validación mediante fstat()
int fd = open("archivo.txt", O_RDONLY | O_NOFOLLOW);
if (fd != -1) {
  struct stat st;
  if (fstat(fd, &st) == 0) {
    /* uso seguro del descriptor */
  }
}
```

- Restringir la resolución de enlaces simbólicos usando `openat()` con un directorio seguro previamente validado.

```
//-- Construcción manual de ruta concatenando directorios
char ruta[256];
snprintf(ruta, sizeof(ruta), "/dir_seguro/%s", nombre);
int fd = open(ruta, O_RDONLY);
```

```
//-- Apertura relativa a descriptor de directorio confiable
int dirfd = open("/dir_seguro", O_DIRECTORY | O_RDONLY);
if (dirfd != -1) {
  int fd = openat(dirfd, nombre, O_RDONLY | O_NOFOLLOW);
```

```
}
```

2.13.2.2 Estándares metodológicos

- **CERT/C-POS35**: Avoid race conditions while checking for the existence of a symbolic link.

2.13.3 [c.posix.3] - Evitar condiciones de carrera al usar fork y descriptores de archivo

ID	c.posix.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los procesos padre e hijo creados con `fork()` comparten los mismos descriptores de archivo, lo que puede generar **condiciones de carrera** cuando ambos procesos intentan acceder simultáneamente a los mismos recursos. Esta situación produce comportamientos impredecibles en operaciones de lectura, escritura y posicionamiento del puntero de archivo, comprometiendo la integridad de los datos y la consistencia del programa.

Codificación insegura

Se muestra un programa que crea un proceso hijo con `fork()` y ambos procesos escriben en el mismo descriptor de archivo sin coordinación. El objetivo es observar cómo el uso compartido de descriptores sin sincronización puede provocar condiciones de carrera y resultados impredecibles en el archivo de salida.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

//-- Demuestra escritura concurrente sin sincronización en un descriptor compartido
int main() {
  int fd = open("salida.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
  if (fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  pid_t pid = fork();
  if (pid < 0) {
    perror("fork");
    close(fd);
    exit(EXIT_FAILURE);
```

```
  }
  //-- Ambos procesos usan el mismo descriptor sin control
  for (int i = 0; i < 5; i++) {
    if (pid == 0) {
      write(fd, "Hijo escribiendo\n", 18);
    } else {
      write(fd, "Padre escribiendo\n", 19);
    }
  }
  if (pid != 0) {
    wait(NULL);
  }
  close(fd);
  return 0;
}
```

Codificación recomendada

Se presenta un programa que crea un proceso hijo con `fork()` y gestiona el acceso al archivo para evitar condiciones de carrera. Se aprende a separar el uso del descriptor o sincronizar los procesos para garantizar un comportamiento predecible y seguro al escribir en recursos compartidos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

//-- Demuestra control del acceso al archivo tras fork
int main() {
  int fd = open("salida_segura.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
  if (fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  pid_t pid = fork();
  if (pid < 0) {
    perror("fork");
    close(fd);
    exit(EXIT_FAILURE);
  }
  if (pid == 0) {
    //-- El hijo escribe y cierra su descriptor inmediatamente
    for (int i = 0; i < 5; i++) {
      write(fd, "Hijo escribiendo\n", 18);
    }
    close(fd);
```

```
    exit(EXIT_SUCCESS);
  } else {
    //-- El padre espera a que el hijo termine antes de escribir
    wait(NULL);
    for (int i = 0; i < 5; i++) {
      write(fd, "Padre escribiendo\n", 19);
    }
    close(fd);
  }
  return 0;
}
```

2.13.3.1 Descripción

El problema surge porque ambos procesos comparten el mismo **offset de archivo** y pueden realizar operaciones de lectura o escritura simultáneamente. Esto genera comportamientos impredecibles donde las operaciones pueden intercalarse de manera no determinista, corrompiendo datos o produciendo resultados inconsistentes.

Los riesgos incluyen **corrupción de datos** cuando múltiples procesos escriben simultáneamente, **lecturas inconsistentes** donde cada proceso obtiene fragmentos parciales de información, y **comportamiento no determinista** que dificulta la depuración y el mantenimiento del código.

La mitigación requiere establecer una **separación clara de responsabilidades** entre procesos respecto al manejo de descriptores de archivo. Esto incluye cerrar descriptores innecesarios inmediatamente después del `fork()`, implementar mecanismos de sincronización cuando el acceso compartido es inevitable, o redirigir los procesos hacia recursos independientes.

La gestión adecuada también implica considerar el **momento del cierre** de descriptores, ya que cerrar un descriptor en un proceso afecta a todos los procesos que lo comparten. La planificación cuidadosa de qué proceso es responsable de cada recurso previene interferencias mutuas y garantiza un comportamiento predecible del programa.

Riesgos asociados

- Divulgación inadvertida de descriptores de archivo a procesos hijos que no deberían tener acceso.
- Pérdida de datos debida a escrituras concurrentes o cierres accidentales de descriptores compartidos.
- Ejecución de código inconsistente debido a estados de descriptores inesperados.
- Elevación de privilegios si el proceso hijo utiliza descriptores con permisos privilegiados.

- Condiciones de carrera explotables que pueden ser utilizadas para ataques de denegación de servicio o corrupción de datos.
- Fugas de recursos por descriptores no cerrados correctamente después del fork.
- Fallos en la aplicación por uso de descriptores de archivo en estados no definidos.

Buenas prácticas

- Cerrar los descriptores de archivo que no se utilicen en cada proceso después de `fork()` para evitar accesos concurrentes no controlados.

```
//-- Ambos procesos usan el mismo descriptor sin cerrarlo
int fd = open("datos.txt", O_WRONLY | O_CREAT, 0644);
pid_t pid = fork();
write(fd, "Padre\n", 6);
if (pid == 0) {
  write(fd, "Hijo\n", 5);
}
```

```
//-- Cada proceso cierra el descriptor que no necesita
int fd = open("datos.txt", O_WRONLY | O_CREAT, 0644);
pid_t pid = fork();
if (pid == 0) {
  write(fd, "Hijo\n", 5);
  close(fd);
} else {
  write(fd, "Padre\n", 6);
  close(fd);
}
```

- Utilizar `O_CLOEXEC` o configurar `FD_CLOEXEC` para evitar la herencia no intencionada de descriptores en procesos ejecutados con `exec()`.

```
//-- El descriptor se hereda tras exec()
int fd = open("secreto.txt", O_RDONLY);
if (fork() == 0) {
  execl("/bin/ls", "ls", NULL);
}
```

```
//-- El descriptor se marca para cerrarse automáticamente en exec()
int fd = open("secreto.txt", O_RDONLY | O_CLOEXEC);
if (fork() == 0) {
  execl("/bin/ls", "ls", NULL);
}
```

- Sincronizar explícitamente los procesos (por ejemplo, con `wait()`) cuando compartan descriptores para evitar escrituras simultáneas no controladas.

```
//-- Padre e hijo escriben sin coordinación
int fd = open("log.txt", O_WRONLY | O_CREAT, 0644);
pid_t pid = fork();
if (pid == 0) {
  write(fd, "Hijo\n", 5);
} else {
  write(fd, "Padre\n", 6);
}
```

```
//-- El padre espera al hijo antes de escribir
int fd = open("log.txt", O_WRONLY | O_CREAT, 0644);
pid_t pid = fork();
if (pid == 0) {
  write(fd, "Hijo\n", 5);
  close(fd);
} else {
  wait(NULL);
  write(fd, "Padre\n", 6);
  close(fd);
}
```

- Usar `O_APPEND` al abrir archivos compartidos para evitar condiciones de carrera en el desplazamiento del puntero de archivo.

```
//-- Escrituras concurrentes sin O_APPEND
int fd = open("registro.txt", O_WRONLY | O_CREAT, 0644);
if (fork() == 0) {
  lseek(fd, 0, SEEK_END);
  write(fd, "Entrada hijo\n", 13);
} else {
  lseek(fd, 0, SEEK_END);
  write(fd, "Entrada padre\n", 14);
}
```

```
//-- Escrituras atómicas al final del archivo con O_APPEND
int fd = open("registro.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
if (fork() == 0) {
  write(fd, "Entrada hijo\n", 13);
} else {
  write(fd, "Entrada padre\n", 14);
}
```

2.13.3.2 Estándares metodológicos

- **CERT/C-POS38**: Beware of race conditions when using fork and file descriptors.

2.13.4 [c.posix.4] - No usar señales para terminar hilos

ID	c.posix.4	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Los hilos de ejecución pueden terminar de forma abrupta cuando reciben ciertas señales del sistema operativo, interrumpiendo operaciones críticas y dejando recursos en estados inconsistentes. El uso de señales como `SIGKILL` o `SIGTERM` para finalizar hilos puede provocar **condiciones de carrera**, **bloqueos de memoria** y **corrupción de datos compartidos**, ya que no permiten una terminación controlada que libere adecuadamente los recursos adquiridos.

Codificación insegura

Uso de señales para intentar detener un hilo en ejecución. Se muestra cómo un hilo es terminado mediante el envío de una señal POSIX, lo que introduce condiciones de carrera y estados inconsistentes. El objetivo es entender por qué este mecanismo no es adecuado para controlar el ciclo de vida de los hilos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>

pthread_t worker;

void handler(int sig) {
  //-- Manejador de señal que finaliza el hilo abruptamente
  printf("Hilo terminado por señal\n");
  pthread_exit(NULL);
}
void* thread_function(void* arg) {
  //-- Registro del manejador de señal dentro del hilo
  signal(SIGTERM, handler);

  //-- Bucle infinito simulando trabajo continuo
  while (1) {
    printf("Trabajando...\n");
    sleep(1);
  }
  return NULL;
```

```
}
int main() {
  //-- Creación del hilo de trabajo
  if (pthread_create(&worker, NULL, thread_function, NULL) != 0) {
    perror("Error creando hilo");
    return EXIT_FAILURE;
  }
  sleep(3);

  //-- Envío de señal para forzar la terminación del hilo
  pthread_kill(worker, SIGTERM);

  pthread_join(worker, NULL);
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Uso de un mecanismo de finalización controlado mediante una variable compartida y sincronizada para indicar al hilo que debe terminar. Se muestra cómo gestionar correctamente el ciclo de vida del hilo evitando el uso de señales y asegurando una terminación ordenada.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>

pthread_t worker;
atomic_int stop = 0;

void* thread_function(void* arg) {
  //-- Bucle controlado mediante bandera atómica
  while (!atomic_load(&stop)) {
    printf("Trabajando...\n");
    sleep(1);
  }
  //-- Liberación ordenada antes de terminar
  printf("Hilo finalizando de forma controlada\n");
  return NULL;
}
int main() {
  //-- Creación del hilo de trabajo
  if (pthread_create(&worker, NULL, thread_function, NULL) != 0) {
    perror("Error creando hilo");
    return EXIT_FAILURE;
  }
```

```
  sleep(3);

  //-- Señalización segura para detener el hilo
  atomic_store(&stop, 1);

  //-- Espera a que el hilo termine correctamente
  pthread_join(worker, NULL);

  return EXIT_SUCCESS;
}
```

2.13.4.1 Descripción

Los hilos de ejecución requieren mecanismos de terminación controlada que preserven la integridad del programa y permitan la liberación ordenada de recursos. El uso de señales para terminar hilos introduce comportamientos impredecibles y puede comprometer la estabilidad del sistema.

Las señales operan a nivel de proceso y su entrega a hilos específicos no está garantizada. El sistema puede dirigir una señal a cualquier hilo del proceso que no la tenga bloqueada, lo que genera **condiciones de carrera** y comportamientos no deterministas. Esta impredecibilidad se agrava cuando múltiples hilos manejan las mismas señales.

La terminación abrupta mediante señales impide que los hilos ejecuten sus rutinas de limpieza. Los **recursos asignados** como memoria dinámica, descriptores de archivo, `mutex` y otros objetos de sincronización pueden quedar sin liberar, causando fugas de memoria y bloqueos permanentes que afectan al proceso completo.

Los hilos interrumpidos por señales pueden dejar **estructuras de datos en estados inconsistentes**. Si un hilo está modificando una estructura compartida cuando recibe una señal de terminación, otros hilos pueden acceder a datos parcialmente actualizados o corruptos.

Las funciones de manejo de señales tienen restricciones sobre qué operaciones pueden realizar de forma segura. Muchas funciones de biblioteca no son **async-signal-safe**, lo que limita las opciones de limpieza durante el manejo de señales y puede introducir comportamientos indefinidos.

La mitigación efectiva implica implementar mecanismos de terminación cooperativa donde los hilos verifican periódicamente condiciones de salida y ejecutan rutinas de limpieza antes de terminar. Los objetos de sincronización como variables de condición y semáforos proporcionan alternativas más controladas para coordinar la terminación entre hilos.

Riesgos asociados

- Condiciones de carrera: el uso de señales para terminar hilos puede interrumpir la ejecución en puntos impredecibles, provocando condiciones de carrera y comportamiento indefinido.
- Corrupción de datos: finalizar hilos abruptamente con señales puede dejar estructuras de datos en estados inconsistente, provocando corrupción de memoria o datos.
- Fugas de memoria: el hilo puede no liberar adecuadamente recursos asignados, resultando en fugas de memoria.
- Bloqueos (deadlocks): si un hilo es terminado mientras sostiene recursos compartidos o locks, puede ocurrir un bloqueo del sistema.
- Violaciones de seguridad: el estado inconsistente puede ser aprovechado para explotar vulnerabilidades de seguridad.
- Pérdida de información: las tareas importantes ejecutadas por el hilo (como escrituras o cierres de archivos) pueden quedar incompletas, llevando a la pérdida de datos.
- Dificultad de depuración: los problemas derivados de terminar hilos mediante señales suelen ser difíciles de identificar y resolver.
- Comportamiento indefinido: la especificación estándar de C no define cómo responde el hilo ante señales inesperadas, lo que puede causar fallos imprevisibles en el programa.

Buenas prácticas

- Utilizar variables de control compartidas y mecanismos de sincronización en lugar de señales para solicitar la finalización de un hilo.

```
//-- Terminación de hilo mediante señal externa (SIGTERM)
void handler(int sig) {
  pthread_exit(NULL);
}
```

```
//-- Terminación controlada mediante bandera compartida protegida
volatile sig_atomic_t stop = 0;
void *worker(void *arg) {
  while (!stop) {
    /* trabajo */
  }
  return NULL;
}
stop = 1;
```

```
pthread_join(thread, NULL);
```

- Emplear variables de condición para notificar a los hilos que deben finalizar, evitando interrupciones asíncronas con señales.

```
//-- Finalización forzada del hilo usando pthread_kill
pthread_kill(thread, SIGUSR1);
```

```
//-- Notificación segura usando mutex y variable de condición
pthread_mutex_lock(&mutex);
terminate = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
void *worker(void *arg) {
  pthread_mutex_lock(&mutex);
  while (!terminate) {
    pthread_cond_wait(&cond, &mutex);
  }
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

- Implementar cancelación diferida con puntos de cancelación controlados en lugar de usar señales del sistema.

```
//-- Uso de señal para interrumpir ejecución bloqueante
kill(getpid(), SIGINT);
```

```
//-- Cancelación diferida configurada explícitamente
void *worker(void *arg) {
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
  while (1) {
    pthread_testcancel();
    /* trabajo */
  }
}
pthread_cancel(thread);
pthread_join(thread, NULL);
```

- Centralizar la gestión de finalización en el hilo principal mediante coordinación explícita, evitando depender de manejadores de señales asíncronos.

```
//-- Registro de manejador de señal para cerrar hilos abruptamente
signal(SIGTERM, handler);
```

```
//-- Coordinación explícita desde el hilo principal
for (int i = 0; i < num_threads; i++) {
  flags[i] = 1;
}
for (int i = 0; i < num_threads; i++) {
  pthread_join(threads[i], NULL);
}
```

2.13.4.2 Estándares metodológicos

- **CERT/C-POS44**: Do not use signals to terminate threads.

2.13.5 [c.posix.5] - No usar hilos cancelables asincrónicamente

ID	c.posix.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los hilos cancelables asincrónicamente pueden interrumpirse en cualquier punto de ejecución, incluso durante operaciones críticas como la liberación de memoria o el desbloqueo de `mutex`. Esta cancelación impredecible genera **condiciones de carrera**, **corrupción de datos** y **bloqueos permanentes** cuando los recursos quedan en estados inconsistentes al no completarse las operaciones de limpieza necesarias.

Codificación insegura

Se muestra la creación de un hilo POSIX configurado con cancelación asíncrona. El objetivo es observar cómo la finalización inmediata puede interrumpir operaciones críticas y dejar recursos en estado inconsistente, lo que representa un riesgo de seguridad y estabilidad.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

//-- Función que simula una sección crítica protegida por mutex
void* worker(void* arg) {
  //-- Se habilita cancelación asíncrona (no seguro)
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

  pthread_mutex_lock(&lock);
  //-- Sección crítica que podría quedar inconsistente
  printf("Hilo trabajando en sección crítica...\n");
```

```
  sleep(5);
  pthread_mutex_unlock(&lock);

  return NULL;
}
//-- Programa principal que cancela el hilo mientras ejecuta la sección crítica
int main() {
  pthread_t tid;

  if (pthread_create(&tid, NULL, worker, NULL) != 0) {
    perror("Error creando hilo");
    return EXIT_FAILURE;
  }
  sleep(1);
  //-- Cancelación inmediata del hilo
  pthread_cancel(tid);

  pthread_join(tid, NULL);
  printf("Hilo cancelado.\n");

  pthread_mutex_destroy(&lock);
  return EXIT_SUCCESS;
}
```

Codificación recomendada

Se presenta el uso de cancelación diferida y puntos de cancelación controlados. El propósito es demostrar cómo permitir la finalización segura del hilo, asegurando la liberación correcta de recursos y evitando estados inconsistentes en secciones críticas.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

//-- Función que usa cancelación diferida y libera recursos correctamente
void cleanup(void* arg) {
  //-- Liberación segura del mutex en caso de cancelación
  pthread_mutex_unlock(&lock);
}
void* worker(void* arg) {
  //-- Se configura cancelación diferida (modo seguro)
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

  pthread_mutex_lock(&lock);
  pthread_cleanup_push(cleanup, NULL);
```

```
  //-- Sección crítica protegida
  printf("Hilo trabajando en sección crítica...\n");
  sleep(5);

  pthread_cleanup_pop(1);
  return NULL;
}
//-- Programa principal que solicita cancelación controlada
int main() {
  pthread_t tid;

  if (pthread_create(&tid, NULL, worker, NULL) != 0) {
    perror("Error creando hilo");
    return EXIT_FAILURE;
  }
  sleep(1);
  //-- Solicitud de cancelación diferida
  pthread_cancel(tid);

  pthread_join(tid, NULL);
  printf("Hilo finalizado de forma segura.\n");

  pthread_mutex_destroy(&lock);
  return EXIT_SUCCESS;
}
```

2.13.5.1 Descripción

Los hilos cancelables asincrónicamente permiten que un hilo sea terminado en cualquier punto de su ejecución sin previo aviso. Esta capacidad de cancelación inmediata genera problemas de consistencia en el estado del programa, ya que el hilo puede ser interrumpido mientras modifica estructuras de datos compartidas, mantiene `mutex` bloqueados o ejecuta operaciones críticas.

La cancelación asincrónica impide que el hilo complete las operaciones en curso y ejecute las rutinas de limpieza necesarias. Cuando un hilo es cancelado de forma abrupta, los recursos que había adquirido permanecen bloqueados, las estructuras de datos pueden quedar en estados inconsistentes y las operaciones atómicas pueden interrumpirse a medias.

Los **puntos de cancelación** definidos en el estándar POSIX proporcionan ubicaciones específicas donde un hilo puede ser cancelado de manera segura. Estas ubicaciones corresponden a funciones del sistema que pueden bloquearse, como operaciones de entrada/salida, donde la cancelación no compromete la integridad del estado del programa.

La cancelación diferida mediante `PTHREAD_CANCEL_DEFERRED` permite que el hilo procese la solicitud de cancelación únicamente cuando alcanza un punto de cancelación. Este mecanismo garantiza que el hilo pueda completar las operaciones críticas y ejecutar las rutinas de limpieza registradas con `pthread_cleanup_push`.

Los riesgos asociados con la cancelación asincrónica incluyen corrupción de datos, interbloqueos permanentes y violaciones de invariantes del programa. Estos problemas son especialmente críticos en aplicaciones multihilo que manejan recursos compartidos o realizan operaciones transaccionales.

La mitigación efectiva requiere el uso exclusivo de cancelación diferida, la implementación de puntos de cancelación explícitos mediante `pthread_testcancel` en bucles largos, y el registro adecuado de rutinas de limpieza para garantizar la liberación de recursos cuando el hilo es cancelado.

Riesgos asociados

- Pérdida de integridad de datos debido a la finalización abrupta de un hilo.
- Condiciones de carrera por recursos compartidos no liberados correctamente.
- Bloqueos (deadlocks) por mantener bloqueos o `mutex` activos cuando el hilo es cancelado.
- Fugas de memoria al no liberar correctamente memoria asignada.
- Corrupción de estructuras de datos si el hilo es cancelado durante una operación crítica.
- Estados inconsistentes en variables globales o compartidas.
- Exposición a vulnerabilidades de denegación de servicio (DoS) por recursos no liberados.
- Difícil identificación y solución de errores por comportamientos no deterministas.
- Incremento en la complejidad del mantenimiento y depuración del código.

Buenas prácticas

- Configurar los hilos para que sean no cancelables o diferir explícitamente la cancelación.

```
//-- Hilo con cancelación asíncrona que puede interrumpir en estado inconsistente
void *worker(void *arg) {
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  procesar_datos_compartidos();
  return NULL;
}
```

```
//-- Hilo con cancelación diferida controlando puntos seguros
void *worker(void *arg) {
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
  while (trabajando) {
    procesar_datos_compartidos();
    pthread_testcancel();
  }
  return NULL;
}
```

- Deshabilitar temporalmente la cancelación en secciones críticas que manipulan recursos compartidos.

```
//-- Sección crítica vulnerable a cancelación asíncrona
pthread_mutex_lock(&mutex);
actualizar_estado_global();
pthread_mutex_unlock(&mutex);
```

```
//-- Cancelación deshabilitada durante sección crítica sensible
int oldstate;
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
pthread_mutex_lock(&mutex);
actualizar_estado_global();
pthread_mutex_unlock(&mutex);
pthread_setcancelstate(oldstate, NULL);
```

- Implementar mecanismos explícitos de finalización cooperativa en lugar de cancelar hilos abruptamente.

```
//-- Cancelación externa directa del hilo
pthread_t tid;
pthread_create(&tid, NULL, worker, NULL);
pthread_cancel(tid);
```

```
//-- Uso de bandera de parada controlada para finalización ordenada
volatile sig_atomic_t detener = 0;
void *worker(void *arg) {
  while (!detener) {
    procesar_tarea();
  }
  liberar_recursos();
  return NULL;
}
```

2.13.5.2 Estándares metodológicos

- **CERT/C-POS47**: Do not use threads that can be canceled asynchronously.

2.14 Números aleatorios

2.14.1 [c.rand.1] - No usar "rand" para generar números pseudoaleatorios

ID	c.rand.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Confidencialidad	S-SDLC	Build

La función `rand()` implementa un generador de números pseudoaleatorios de baja calidad criptográfica que produce secuencias predecibles. Su uso en contextos que requieren **aleatoriedad criptográfica** introduce vulnerabilidades de seguridad, ya que los atacantes pueden predecir los valores generados y comprometer la seguridad del sistema.

Codificación insegura

Generación de un número pseudoaleatorio para un valor entre 0 y 99. Se busca ilustrar cómo obtener un valor aparentemente aleatorio y qué implicaciones tiene la fuente utilizada para generarlo, especialmente cuando puede influir en decisiones de seguridad.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
  //-- Inicializa la semilla con el tiempo actual
  srand((unsigned int)time(NULL));

  //-- Genera un número pseudoaleatorio entre 0 y 99
  int numero = rand() % 100;

  //-- Muestra el número generado
  printf("Numero generado: %d\n", numero);

  return 0;
}
```

Codificación recomendada

Generación de un número pseudoaleatorio entre 0 y 99 utilizando una fuente de entropía del sistema operativo. Se pretende mostrar cómo obtener valores menos

predecibles cuando el número puede utilizarse en contextos sensibles como tokens, identificadores o elementos relacionados con seguridad.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(void) {
  //-- Abre la fuente de entropía del sistema operativo
  FILE *fp = fopen("/dev/urandom", "rb");
  if (fp == NULL) {
    //-- Maneja el error si no se puede acceder a la fuente
    perror("No se pudo abrir /dev/urandom");
    return EXIT_FAILURE;
  }
  //-- Lee un entero de 32 bits desde la fuente segura
  uint32_t valor;
  if (fread(&valor, sizeof(valor), 1, fp) != 1) {
    //-- Maneja el error si la lectura falla
    perror("Error al leer bytes aleatorios");
    fclose(fp);
    return EXIT_FAILURE;
  }
  //-- Cierra la fuente de entropía
  fclose(fp);

  //-- Ajusta el valor al rango deseado
  int numero = valor % 100;

  //-- Muestra el número generado
  printf("Numero generado: %d\n", numero);

  return EXIT_SUCCESS;
}
```

En Windows, esta misma solución sería:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <windows.h>
#include <bcrypt.h>

#pragma comment(lib, "bcrypt.lib")

int main(void) {
  uint32_t valor;
  //-- Obtiene bytes aleatorios desde el generador criptográfico del sistema
operativo
```

```
    NTSTATUS status = BCryptGenRandom(
        NULL,
        (PUCHAR)&valor,
        sizeof(valor),
        BCRYPT_USE_SYSTEM_PREFERRED_RNG
    );
    if (status != 0) {
      //-- Maneja el error si no se pudieron obtener bytes aleatorios
      fprintf(stderr, "Error al generar bytes aleatorios\n");
      return EXIT_FAILURE;
    }
    //-- Ajusta el valor al rango deseado
    int numero = valor % 100;

    //-- Muestra el número generado
    printf("Numero generado: %d\n", numero);

    return EXIT_SUCCESS;
}
```

2.14.1.1 Descripción

La función `rand()` produce secuencias predecibles que pueden ser reproducidas por atacantes que conozcan el estado interno o la semilla utilizada, y por tanto, no debería utilizarse en contextos donde se requiere seguridad.

El algoritmo subyacente de `rand()` está diseñado para aplicaciones generales como simulaciones o juegos, no para generar material criptográfico. Su **entropía limitada** y la **predictibilidad de sus secuencias** comprometen cualquier mecanismo de seguridad que dependa de la aleatoriedad, como la generación de claves, tokens de autenticación, nonces criptográficos o identificadores de sesión.

Los riesgos asociados incluyen la posibilidad de que atacantes predigan valores futuros, reproduzcan secuencias completas o realicen ataques de fuerza bruta más eficientes contra sistemas que dependan de estos valores pseudoaleatorios para su seguridad.

La mitigación requiere el uso de **generadores criptográficamente seguros** proporcionados por el sistema operativo o bibliotecas especializadas. Estos generadores obtienen entropía de fuentes hardware o del kernel del sistema operativo, garantizando que las secuencias generadas sean impredecibles incluso para atacantes con conocimiento parcial del estado del sistema.

En sistemas que requieren compatibilidad con estándares específicos, debe verificarse que los generadores utilizados cumplan con los requisitos de aleatoriedad establecidos por organismos como NIST o FIPS para aplicaciones criptográficas.

Riesgos asociados

- Predicción fácil de números aleatorios por parte de un atacante.
- Vulnerabilidad a ataques de adivinación de valores generados.
- Compromiso de mecanismos de autenticación, tokens o sesiones si dependen de valores generados por `rand()`.
- Posibilidad de crear patrones repetibles debido a la baja entropía de `rand()`.
- Debilidad ante ataques de repetición o suplantación.
- Falta de cumplimiento de estándares de seguridad (por ejemplo, PCI-DSS exige generadores criptográficamente seguros).
- Exposición a ataques de fuerza bruta por la previsibilidad del algoritmo.
- Falsas garantías de seguridad al utilizar `rand()` en contextos críticos.

Buenas prácticas

- Utilizar generadores de números aleatorios criptográficamente seguros en lugar de `rand()` cuando la aleatoriedad influya en la seguridad.

```
//-- Generación de token con rand predecible
char token[16];
for (int i = 0; i < 16; i++) {
  token[i] = 'A' + (rand() % 26);
}
```

```
//-- Generación de token usando fuente criptográficamente segura
char token[16];
FILE *fp = fopen("/dev/urandom", "rb");
if (fp == NULL) {
  if (fread(token, 1, sizeof(token), fp) != sizeof(token)) {
    fclose(fp);
    return -1; //-- gestionar error.
  }
  fclose(fp);
}
```

- Evitar sembrar `rand()` con valores predecibles como `time(NULL)` para datos sensibles.

```
//-- Inicialización predecible basada en el tiempo
srand(time(NULL));
int value = rand();
```

```
//-- Uso de API segura del sistema para obtener entero aleatorio
```

```
uint32_t value;
FILE *fp = fopen("/dev/urandom", "rb");
if (fp != NULL) {
  if (fread(&value, sizeof(value), 1, fp) != 1) {
    fclose(fp);
    return -1; //-- gestionar error.
  }
  fclose(fp);
}
```

- Emplear APIs específicas del sistema como `getrandom()` o equivalentes en lugar de `rand()` para claves o identificadores.

```
//-- Generación de identificador usando rand limitado por módulo
int id = rand() % 100000;
```

```
//-- Generación de identificador usando getrandom
#include <sys/random.h>
uint32_t id;
if (getrandom(&id, sizeof(id), 0) == sizeof(id)) {
  id = id % 100000;
}
```

- Utilizar bibliotecas criptográficas consolidadas (por ejemplo, OpenSSL) para generar bytes aleatorios seguros.

```
//-- Clave generada con rand byte a byte
unsigned char key[32];
for (int i = 0; i < 32; i++) {
  key[i] = rand() & 0xFF;
}
```

```
//-- Clave generada con RAND_bytes de OpenSSL
unsigned char key[32];
if (RAND_bytes(key, sizeof(key)) == 1) {
  // uso de la clave generada
}
```

2.14.1.2 Estándares metodológicos

- **CERT/C-MSC30**: Do not use the rand() function for generating pseudorandom numbers.

2.14.2 [c.rand.2] - Inicializar correctamente el generador de números aleatorios

ID	c.rand.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Confidencialidad	S-SDLC	Build

Los generadores de números aleatorios requieren una **inicialización adecuada** para producir secuencias impredecibles. Sin una semilla apropiada, las funciones como `rand()` generan **patrones predecibles** que comprometen la seguridad de aplicaciones criptográficas, autenticación y cualquier funcionalidad que dependa de la aleatoriedad.

Codificación insegura

Generación de números aleatorios sin una inicialización adecuada del generador, lo que provoca que la secuencia sea predecible en cada ejecución. Se pretende mostrar cómo la falta de una semilla variable afecta a la seguridad y a la imprevisibilidad de los valores generados.

```
#include <stdio.h>
#include <stdlib.h>

//-- Programa que genera números aleatorios sin inicializar correctamente la
semilla
int main(void) {
  int i;

  //-- No se llama a srand(), se usa la semilla por defecto
  for (i = 0; i < 5; i++) {
    int numero = rand() % 100;
    printf("Numero generado: %d\n", numero);
  }
  return 0;
}
```

Codificación recomendada

Generación de números aleatorios inicializando correctamente el generador con una semilla basada en el tiempo actual, logrando que la secuencia sea diferente en cada ejecución. Se busca aprender cómo reducir la predictibilidad en el uso básico de rand().

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>
```

```
int main(void) {
  struct timespec ts;
  unsigned int seed;
  int i;
  if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
    return -1; //-- gestionar error.
  }
  seed = (unsigned int)(ts.tv_sec ^ ts.tv_nsec);
  srand(seed);
  for (i = 0; i < 5; i++) {
    int numero = rand() % 100;
    printf("Numero generado: %d\n", numero);
  }
  return 0;
}
```

2.14.2.1 Descripción

La función `rand()` genera números pseudoaleatorios basados en un algoritmo determinista que utiliza un valor inicial denominado semilla. Si no se establece explícitamente esta semilla mediante `srand()`, el generador utiliza un valor por defecto fijo, produciendo la misma secuencia de números en cada ejecución del programa.

Esta predictibilidad representa un **riesgo significativo para la seguridad** cuando los números aleatorios se utilizan en operaciones criptográficas, generación de tokens de autenticación, claves de sesión o cualquier contexto donde la impredecibilidad sea necesaria para mantener la seguridad del sistema.

La inicialización debe realizarse **una sola vez** durante la ejecución del programa, preferiblemente al inicio. Llamadas múltiples a `srand()` con la misma semilla reinician el generador al mismo estado, eliminando la aleatoriedad acumulada.

Para mitigar estos riesgos, se debe inicializar el generador con una semilla que varíe entre ejecuciones. Las fuentes comunes incluyen el tiempo actual del sistema, aunque para aplicaciones que requieren alta seguridad criptográfica, se recomienda utilizar generadores específicamente diseñados para criptografía que ofrecen mayor entropía y resistencia a ataques.

Es importante considerar que `rand()` no está diseñado para uso criptográfico y puede presentar patrones estadísticos predecibles. En contextos de seguridad crítica, se deben emplear generadores criptográficamente seguros proporcionados por bibliotecas especializadas o APIs del sistema operativo.

Riesgos asociados

- Predicción de valores aleatorios, facilitando ataques como la suplantación de identidad o el descifrado de datos.

- Repetición de secuencias aleatorias, lo que puede provocar que los mismos valores se utilicen en diferentes sesiones.
- Vulnerabilidad ante ataques de fuerza bruta por secuencias poco aleatorias o fácilmente deducibles.
- Generación de contraseñas, tokens o claves poco seguras.
- Fallos en algoritmos criptográficos que dependan de entropía adecuada.
- Compromiso de la integridad, confidencialidad y autenticidad en aplicaciones de seguridad.
- Debilidad en procesos de inicialización de protocolos o sesiones seguras.
- Exposición de patrones de comportamiento en sistemas transaccionales o de juegos.

Buenas prácticas

- Inicializar el generador de números aleatorios una sola vez utilizando una semilla basada en una fuente de entropía adecuada.

```
//-- Semilla constante predecible
srand(1);
int valor = rand();
```

```
//-- Semilla basada en el tiempo actual
srand((unsigned int)time(NULL));
int valor = rand();
```

- Evitar reinicializar el generador en cada llamada a una función.

```
void generar() {
  //-- Reinicialización en cada invocación
  srand((unsigned int)time(NULL));
  int v = rand();
}
```

```
void generar() {
  //-- Uso del generador previamente inicializado
  int v = rand();
}
```

- Para contextos sensibles a la seguridad, utilizar fuentes de aleatoriedad criptográficamente seguras.

```
//-- Uso de rand() para token sensible
```

```
srand((unsigned int)time(NULL));
int token = rand();
```

```
//-- Uso de generador criptográficamente seguro
unsigned int token;
getrandom(&token, sizeof(token), 0);
```

- Verificar que la semilla tenga suficiente variabilidad en sistemas concurrentes o de alta frecuencia.

```
//-- Posible repetición de semilla en ejecuciones rápidas
srand((unsigned int)time(NULL));
int id = rand();
```

```
//-- Combinación de tiempo y PID para mayor variabilidad
srand((unsigned int)(time(NULL) ^ getpid()));
int id = rand();
```

- Encapsular la inicialización del generador en un módulo controlado para evitar usos indebidos.

```
//-- Inicialización dispersa en múltiples archivos
srand((unsigned int)time(NULL));
int a = rand();
```

```
//-- Inicialización centralizada en función dedicada
void init_random() {
  srand((unsigned int)time(NULL));
}
int a = rand();
```

2.14.2.2 Estándares metodológicos

- **CERT/C-MSC32**: Properly seed pseudorandom number generators.

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