Guía de Desarrollo Seguro
CCN-STIC 481 – Anexo D

Codificación Segura para Rust



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 Rust API Guidelines 2
1.3.2 Rustonomicon 2
1.4 ESTRUCTURA DE LA GUÍA 3

2. REGLAS DE CODIFICACIÓN SEGURA 5

2.1 INICIALIZACIÓN 5
2.1.1 [rust.ini.1] - No exponer valores en estados parcial o temporalmente inválidos fuera de constructores 5
2.1.2 [rust.ini.2] - No usar mem::zeroed ni MaybeUninit salvo siguiendo patrones canónicos documentados 10
2.1.3 [rust.ini.3] - Garantizar que todo valor observable cumple invariantes de tipo en todo momento 14
2.2 GESTIÓN DE MEMORIA 19
2.2.1 [rust.mem.1] - No introducir bloques unsafe sin invariantes de seguridad explícitos, completos y verificables 19
2.2.2 [rust.mem.2] - No asumir aliasing válido fuera de las garantías del sistema de tipos 23
2.2.3 [rust.mem.3] - No depender del layout de memoria sin usar repr(C), repr(transparent) o repr(packed) cuando proceda 28
2.2.4 [rust.mem.4] - No realizar accesos fuera de límites en arrays, slices o regiones apuntadas por punteros 32
2.2.5 [rust.mem.5] - No permitir uso-after-free, doble liberación ni referencias colgantes 36
2.2.6 [rust.mem.6] - No leer memoria no inicializada ni parcialmente inicializada 40
2.3 GESTIÓN DE RECURSOS 44
2.3.1 [rust.res.1] - No depender de orden implícito de drop para mantener invariantes críticos 44
2.3.2 [rust.res.2] - Liberar recursos externos de forma determinista mediante RAII 50
2.3.3 [rust.res.3] - No filtrar recursos en rutas de error, panic o cancelación 55
2.4 VALIDACIONES 60
2.4.1 [rust.val.1] - No deserializar directamente datos no confiables a tipos con invariantes complejos 60
2.4.2 [rust.val.2] - Validar tamaños y longitudes antes de realizar asignaciones dependientes de entrada externa 65
2.4.3 [rust.val.3] - No confiar en derives automáticos cuando puedan violar invariantes de seguridad 70
2.5 GESTIÓN DE ERRORES 74
2.5.1 [rust.err.1] - No usar unwrap, expect o panic en código productivo donde el fallo sea posible 74
2.5.2 [rust.err.2] - Representar fallos recuperables exclusivamente mediante Result u Option . 77
2.5.3 [rust.err.3] - No ignorar valores Result marcados como must_use 81
2.5.4 [rust.err.4] - No permitir que un panic cruce límites FFI 85
2.5.5 [rust.err.5] - Garantizar que Drop mantiene invariantes incluso ante panic 89
2.5.6 [rust.err.6] - No depender de ejecución completa de destructores para mantener invariantes críticos 94
2.6 CONCURRENCIA 99
2.6.1 [rust.sync.1] - No compartir estado mutable entre hilos sin sincronización garantizada por Send y Sync 99

2.6.2 [rust.sync.2] - No introducir data races mediante interior mutability sin justificación formal 104
2.6.3 [rust.sync.3] - No asumir orden de ejecución ni visibilidad de memoria sin primitivas de sincronización 108
2.6.4 [rust.sync.4] - No usar ordenamientos Relaxed sin justificación formal basada en el modelo de memoria 113
2.6.5 [rust.sync.5] - No mezclar atomics y locks sin un modelo de sincronización coherente ... 118
2.6.6 [rust.sync.6] - No asumir ejecución completa de futures ante cancelación 123
2.6.7 [rust.sync.7] - No mantener invariantes críticas a través de puntos await 129
2.6.8 [rust.sync.8] - Garantizar liberación de recursos aunque una tarea sea cancelada 134
2.7 USO DE UNSAFE Y FFI 139
2.7.1 [rust.uns.1] - Encapsular todo código unsafe dentro de abstracciones seguras con API pública libre de unsafe 139
2.7.2 [rust.uns.2] - Minimizar el tamaño, alcance y frecuencia de cada bloque unsafe 143
2.7.3 [rust.uns.3] - Validar exhaustivamente toda entrada proveniente de FFI antes de su uso 147
2.7.4 [rust.uns.4] - No asumir inicialización, alineación ni validez de memoria recibida desde código externo 152

3. ANEXO: METODOLOGÍA DE CÁLCULO DEL SCORE CWSS 157

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 Rust API Guidelines

https://rust-lang.github.io/api-guidelines/checklist.html

Son un conjunto de recomendaciones elaboradas principalmente por el equipo de bibliotecas de Rust, destilando la experiencia acumulada construyendo la librería estándar y el ecosistema de crates. Seguirlas es voluntario, pero hacerlo marca la diferencia entre una librería que encaja naturalmente en el ecosistema y una que genera fricción.

Su contenido abarca convenciones de nombrado, cómo prefixar conversiones, nombrar getters sin el prefijo `get_`, o estructurar los features de Cargo. También incluye contenido en materia de interoperabilidad, donde se insiste en implementar proactivamente los traits comunes de `std` para evitar que los usuarios de tu crate queden bloqueados por el sistema de orphans de Rust. También cubren documentación, licencias y otras necesidades prácticas de cualquier crate publicado.

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

- **Rust-5.CTOR**: Constructors are static, inherent methods.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rust-8.DTOR-FAIL**: Destructors never fail.

1.3.2 Rustonomicon

https://doc.rust-lang.org/nomicon/

El Rustonomicon es una guía avanzada sobre **Unsafe Rust**, orientada a programadores que ya dominan Rust y sistemas de bajo nivel. Su propósito es explicar todo lo que necesitas saber para escribir código inseguro de forma correcta: al adentrarte en el lado oscuro del lenguaje, entiendes mucho mejor el lado de la luz. Comprendes por qué el borrow checker existe, por qué ciertas abstracciones están diseñadas como están, y por qué las soluciones seguras de Rust son seguras.

Cubre temas como memoria no inicializada, subtipado y varianza, seguridad ante panics, concurrencia, FFI (interoperabilidad con otros lenguajes), type punning y cómo el código se traduce a primitivas del compilador, SO y hardware.

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

- **Rustonomicon 6.1**: Constructors.
- **Rustonomicon 7.1**: Exception Safety.
- **Rustonomicon 12.1**: #[panic_handler].

1.4 Estructura de la guía

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

- Inicialización.
- Gestión de memoria.
- Gestión de recursos.
- Validaciones.
- Gestión de errores.
- Concurrencia.
- Uso de unsafe y FFI.

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 Inicialización

2.1.1 [rust.ini.1] - No exponer valores en estados parcial o temporalmente inválidos fuera de constructores

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

Los constructores pueden contener estados temporalmente inválidos durante su ejecución, pero estos estados no deben ser accesibles desde el exterior. La exposición de valores en estados parciales puede permitir que otros componentes del programa accedan a datos inconsistentes o incompletos, generando comportamientos impredecibles y vulnerabilidades de seguridad.

Codificación insegura

Se muestra una estructura que puede quedar en un estado inválido después de su creación, permitiendo que otras partes del código accedan a datos incompletos o inconsistentes. La intención es evidenciar cómo exponer campos públicos y permitir modificaciones posteriores puede dejar el objeto en un estado temporal no válido.

```
struct User {
  //-- Campos públicos que pueden quedar en estado inválido
  pub username: String,
  pub email: String,
}
impl User {
  //-- Constructor que no garantiza un estado completamente válido
  fn new(username: String) -> Self {
    User {
      username,
      email: String::new(), // email vacío (estado inválido)
    }
  }
}
fn main() {
  //-- Se crea el usuario con email vacío
  let mut user = User::new(String::from("alice"));

  //-- Se accede al estado inválido antes de corregirlo
  println!("Usuario: {}, Email: {}", user.username, user.email);

  //-- Se corrige el estado después de haber sido expuesto
```

```
  user.email = String::from("alice@example.com");

  println!("Usuario: {}, Email: {}", user.username, user.email);
}
```

Codificación recomendada

Se presenta una estructura que garantiza que el objeto solo puede existir en un estado válido. Los campos no se exponen directamente y el constructor valida las invariantes antes de crear la instancia, evitando que el objeto sea visible en un estado parcial o temporalmente inválido.

```
struct User {
  //-- Campos privados para proteger la invariante
  username: String,
  email: String,
}
impl User {
  //-- Constructor que valida completamente el estado antes de crear la instancia
  fn new(username: String, email: String) -> Result {
    if username.trim().is_empty() {
      return Err(String::from("El username no puede estar vacío"));
    }
    if !email.contains("@") {
      return Err(String::from("El email no es válido"));
    }
    Ok(User {
      username,
      email,
    })
  }
  //-- Método de solo lectura controlado
  fn email(&self) -> &str {
    &self.email
  }
}
fn main() {
  //-- Solo se obtiene una instancia si cumple las invariantes
  let user = User::new(
    String::from("alice"),
    String::from("alice@example.com"),
  );
  match user {
    Ok(u) => println!("Email válido: {}", u.email()),
    Err(e) => println!("Error al crear usuario: {}", e),
  }
}
```

2.1.1.1 Descripción

Los riesgos asociados incluyen el acceso a datos no inicializados, la violación de invariantes de tipo, condiciones de carrera en entornos concurrentes, y la posibilidad de que código externo opere sobre datos en estados inconsistentes. Estos problemas pueden derivar en comportamientos impredecibles, corrupción de memoria y vulnerabilidades de seguridad.

El sistema de tipos proporciona mecanismos para encapsular el proceso de construcción y garantizar que solo se expongan valores completamente inicializados y válidos. Los **constructores privados**, los **métodos de construcción controlada** y el uso de tipos que representan estados de inicialización constituyen las principales estrategias de mitigación.

La aplicación de esta regla requiere diseñar cuidadosamente las interfaces de construcción para mantener los invariantes de tipo durante todo el proceso de inicialización. Esto implica controlar la visibilidad de los campos durante la construcción y proporcionar métodos públicos que solo retornen instancias completamente válidas.

El cumplimiento de esta regla es especialmente crítico en tipos que manejan recursos del sistema, estructuras de datos con restricciones de integridad complejas, y componentes que operan en entornos concurrentes donde el acceso a estados parciales puede introducir vulnerabilidades de seguridad.

Riesgos asociados

- Fugas de invariantes que pueden causar comportamiento indefinido.
- Exposición de datos sin validar a otras partes del código, lo que puede llevar a operaciones inseguras.
- Creación de objetos en estados incorrectos, permitiendo explotación de errores lógicos.
- Violación de las garantías de seguridad del sistema de tipos de Rust, generando posibles bugs difíciles de detectar.
- Incremento de vulnerabilidades por abuso de estados intermedios, facilitando ataques como time-of-check-to-time-of-use (TOCTOU).
- Inconsistencias al serializar o registrar información sensible antes de su validación completa.
- Desbordamiento de memoria o acceso a recursos en estados irregulares, provocando denegación de servicio o corrupción de datos.
- Dificultad para realizar validaciones automáticas y mantenibilidad del código al depender de estados temporales visibles.

Buenas prácticas

- Evitar exponer campos públicos que puedan quedar en estados inválidos durante la construcción o modificación; utilizar encapsulación y constructores que garanticen invariantes.

```
//-- Estructura con campo público que puede quedar en estado inválido
pub struct Usuario {
  pub edad: u8,
}

let mut u = Usuario { edad: 0 };
u.edad = 255;
```

```
//-- Estructura con campo privado y validación en constructor
pub struct Usuario {
  edad: u8,
}

impl Usuario {
  pub fn new(edad: u8) -> Result<Self, String> {
    if edad > 120 {
      return Err("Edad inválida".into());
    }
    Ok(Usuario { edad })
  }
}
```

- Evitar patrones de inicialización en múltiples pasos que permitan exponer objetos parcialmente construidos; usar funciones asociadas que construyan el objeto completo.

```
//-- Inicialización en dos fases que deja el objeto en estado temporal inválido
struct Config {
  host: String,
  port: u16,
}

let mut c = Config { host: String::new(), port: 0 };
c.host = "localhost".into();
c.port = 8080;
```

```
//-- Constructor que garantiza estado consistente desde el inicio
impl Config {
  pub fn new(host: String, port: u16) -> Result<Self, String> {
    if port == 0 {
      return Err("Puerto inválido".into());
```

```
    }
    Ok(Config { host, port })
  }
}
```

- No proporcionar métodos públicos que permitan romper invariantes internos; exponer únicamente operaciones que mantengan el estado válido.

```
//-- Método público que permite dejar el estado inconsistente
pub struct Cuenta {
  saldo: i32,
}

impl Cuenta {
  pub fn set_saldo(&mut self, saldo: i32) {
    self.saldo = saldo;
  }
}
```

```
//-- Método que valida y preserva invariantes
impl Cuenta {
  pub fn retirar(&mut self, monto: i32) -> Result<(), String> {
    if monto < 0 || monto > self.saldo {
      return Err("Operación inválida".into());
    }
    self.saldo -= monto;
    Ok(())
  }
}
```

2.1.1.2 Estándares metodológicos

- **Rust-5.CTOR**: Constructors are static, inherent methods.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 6.1**: Constructors.

2.1.2 [rust.ini.2] - No usar mem::zeroed ni MaybeUninit salvo siguiendo patrones canónicos documentados

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

Las funciones `mem::zeroed` y el tipo `MaybeUninit` permiten crear valores no inicializados o inicializados con ceros, pero su uso incorrecto puede generar **comportamiento indefinido**. Estos mecanismos requieren garantizar manualmente que los valores cumplan los invariantes de sus tipos antes de ser utilizados, lo que introduce riesgos de seguridad y errores sutiles en el código.

Codificación insegura

Inicialización manual de una estructura que contiene tipos no triviales usando memoria puesta a cero. Se pretende mostrar los riesgos de crear instancias inválidas que violan las invariantes internas de tipos como `String`, lo que puede provocar comportamiento indefinido.

```
use std::mem;

struct User {
  name: String,
  age: u32,
}
fn main() {
  //-- Se crea una instancia poniendo toda la memoria a cero
  //-- String no permite el estado todo a cero y esto viola sus invariantes
internas
  let user: User = unsafe { mem::zeroed() };

  //-- El uso posterior puede provocar comportamiento indefinido
  println!("Nombre: {}", user.name);
  println!("Edad: {}", user.age);
}
```

Codificación recomendada

Inicialización controlada de una estructura usando `MaybeUninit` siguiendo el patrón canónico: reservar memoria sin inicializar, escribir explícitamente cada campo válido y finalmente convertirla en una instancia completamente inicializada.

```
use std::mem::MaybeUninit;

struct User {
```

```
  name: String,
  age: u32,
}
fn main() {
  //-- Se reserva memoria sin inicializar para la estructura completa
  let mut user_uninit: MaybeUninit<User> = MaybeUninit::uninit();

  //-- Se obtiene un puntero mutable para inicializar los campos manualmente
  let user_ptr = user_uninit.as_mut_ptr();

  unsafe {
    //-- Se escriben campos válidos respetando las invariantes de cada tipo
    (*user_ptr).name = String::from("Alice");
    (*user_ptr).age = 30;

    //-- Una vez todos los campos están correctamente inicializados se asume init
    let user = user_uninit.assume_init();

    println!("Nombre: {}", user.name);
    println!("Edad: {}", user.age);
  }
}
```

2.1.2.1 Descripción

El uso de `mem::zeroed` con tipos que no admiten representaciones de bits cero válidas resulta en comportamiento indefinido inmediato. Muchos tipos, incluyendo referencias, punteros no nulos, enums con discriminantes específicos y tipos con restricciones de validez, no pueden inicializarse de forma segura mediante el llenado con ceros de su representación en memoria.

`MaybeUninit` permite trabajar con memoria no inicializada de forma más controlada, pero exige al programador garantizar manualmente que los datos estén completamente inicializados antes de asumir su validez. La lectura de datos no inicializados o parcialmente inicializados constituye comportamiento indefinido y puede exponer información sensible de la memoria.

Los riesgos asociados incluyen corrupción de memoria, exposición de datos residuales, fallos de segmentación y la posibilidad de que atacantes exploten el comportamiento impredecible resultante. El compilador y las herramientas de análisis estático no pueden detectar todos los usos incorrectos de estas primitivas.

La mitigación principal consiste en **evitar completamente** estas funciones salvo en casos excepcionales donde se sigan patrones bien documentados y validados por la comunidad. Las alternativas seguras incluyen constructores por defecto, inicialización explícita de campos, y el uso de tipos que garantizan inicialización segura. Cuando su

uso sea inevitable, debe limitarse a bibliotecas de bajo nivel con revisiones exhaustivas de código y documentación detallada de los invariantes de seguridad.

Riesgos asociados

- Corrupción de memoria por inicialización incompleta de estructuras o tipos que requieren invariantes internos.
- Fugas de información sensible si las áreas de memoria no se limpian correctamente antes de ser usadas o liberadas.
- Invocación de comportamiento indefinido al crear valores inválidos para tipos que no admiten todos los patrones de bits.
- Posible ejecución de código después de violar la seguridad de tipos (type safety), comprometiendo la integridad del programa.
- Errores sutiles y difíciles de depurar debido a la inicialización incorrecta de valores en tipos que esperan ciertos valores por defecto.
- Fallos de seguridad explotables si vectores como double-free o use-after-free son introducidos por un estado de memoria inválido.
- Desbordamientos o fallos de lectura si el uso indebido de `mem::zeroed` o `MaybeUninit` deja punteros o referencias en estados inválidos.

Buenas prácticas

- Evitar el uso de `mem::zeroed` para inicializar tipos complejos o no triviales; preferir constructores explícitos o la implementación de `Default`.

```
//-- Inicialización forzada con memoria en cero
let s: String = unsafe { std::mem::zeroed() };
```

```
//-- Inicialización mediante constructor seguro
let s: String = String::new();
```

- No utilizar `mem::zeroed` con tipos que no permiten todos los bits en cero como estado válido (por ejemplo, referencias o `NonNull`).

```
//-- Referencia creada con memoria en cero
let r: &i32 = unsafe { std::mem::zeroed() };
```

```
//-- Referencia válida a un valor existente
let value = 10;
let r: &i32 = &value;
```

- Evitar el uso manual de `MaybeUninit` sin seguir el patrón documentado de

inicialización y llamada explícita a `assume_init` únicamente tras garantizar la asignación completa.

```
//-- Uso de MaybeUninit sin inicialización garantizada
let mut data: std::mem::MaybeUninit<i32> = std::mem::MaybeUninit::uninit();
let value = unsafe { data.assume_init() };
```

```
//-- Inicialización explícita antes de asumir el valor
let mut data: std::mem::MaybeUninit<i32> = std::mem::MaybeUninit::uninit();
data.write(42);
let value = unsafe { data.assume_init() };
```

- Para estructuras compuestas o arrays, preferir inicializadores seguros o utilidades estándar en lugar de inicialización parcial con `MaybeUninit` sin control completo del flujo.

```
//-- Array parcialmente inicializado manualmente
let mut arr: [std::mem::MaybeUninit<i32>; 2] = unsafe {
std::mem::MaybeUninit::uninit().assume_init() };
arr[0].write(1);
let arr = unsafe { std::mem::transmute::<_, [i32; 2]>(arr) };
```

```
//-- Inicialización directa y segura del array
let arr = [1, 2];
```

- Encapsular cualquier uso imprescindible de `MaybeUninit` dentro de abstracciones seguras que validen completamente la inicialización antes de exponer el valor.

```
//-- Exposición directa de MaybeUninit sin encapsulación
pub fn crear() -> i32 {
  let data = std::mem::MaybeUninit::uninit();
  unsafe { data.assume_init() }
}
```

```
//-- Encapsulación que garantiza la inicialización
pub fn crear() -> i32 {
  let mut data = std::mem::MaybeUninit::uninit();
  data.write(7);
  unsafe { data.assume_init() }
}
```

2.1.2.2 Estándares metodológicos

- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rustonomicon 2.1**: repr(Rust).

- **Rustonomicon 5.1**: Checked.

2.1.3 [rust.ini.3] - Garantizar que todo valor observable cumple invariantes de tipo en todo momento

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

Los tipos en Rust definen contratos que establecen qué valores son válidos y qué operaciones están permitidas. Cuando estos **invariantes de tipo** se rompen, el programa puede exhibir comportamientos indefinidos, corrupción de memoria o fallos de seguridad. Las violaciones ocurren típicamente mediante el uso incorrecto de `unsafe`, transmutaciones de memoria no válidas, o manipulación directa de representaciones internas que dejan los tipos en estados inconsistentes.

Codificación insegura

Se muestra una estructura que expone directamente sus campos y permite asignar cualquier valor sin validar reglas básicas del dominio, como evitar números negativos en una edad. La intención es observar cómo un valor inválido puede existir y ser observable desde cualquier parte del programa, rompiendo invariantes del tipo.

```
struct User {
  //-- Campo público que puede tomar cualquier valor entero
  pub age: i32,
}
fn main() {
  //-- Se crea un usuario con una edad válida inicialmente
  let mut user = User { age: 30 };

  //-- Se asigna un valor inválido que rompe la regla de negocio (edad negativa)
  user.age = -10;

  //-- Se imprime un estado inválido que ya es observable en el sistema
  println!("Edad del usuario: {}", user.age);
}
```

Codificación recomendada

Se define un tipo que encapsula el valor y evita que estados inválidos puedan crearse u observarse. La construcción del objeto valida la regla de negocio y el campo interno no es accesible directamente, garantizando que el invariante se mantenga durante todo el ciclo de vida del valor.

```
struct Age {
```

```
  //-- Campo privado que no puede modificarse directamente desde fuera
  value: i32,
}
impl Age {
  //-- Constructor que valida el invariante antes de crear la instancia
  fn new(value: i32) -> Result {
    if value < 0 {
      return Err("La edad no puede ser negativa".to_string());
    }
    Ok(Age { value })
  }
  //-- Método público de solo lectura para exponer el valor sin permitir
modificación
  fn value(&self) -> i32 {
    self.value
  }
}
fn main() {
  //-- Intento de crear una edad inválida
  let invalid_age = Age::new(-10);
  println!("Resultado edad inválida: {:?}", invalid_age);
  //-- Creación correcta que cumple el invariante
  let valid_age = Age::new(25).expect("Edad válida");
  println!("Edad válida: {}", valid_age.value());
}
```

2.1.3.1 Descripción

Los invariantes de tipo definen las condiciones que deben mantenerse válidas para que un tipo funcione correctamente. Estos invariantes establecen restricciones sobre los valores que puede contener una estructura de datos y las relaciones que deben preservarse entre sus campos.

La violación de invariantes de tipo puede ocurrir cuando se modifica el estado interno de una estructura sin validar que las nuevas condiciones cumplan las restricciones establecidas. Esto sucede frecuentemente durante operaciones de mutación, deserialización de datos externos, o cuando se exponen campos internos que permiten modificaciones no controladas.

El incumplimiento de invariantes genera comportamientos indefinidos, corrupción de datos, y fallos en tiempo de ejecución que pueden ser explotados por atacantes. Los sistemas que no garantizan la consistencia de sus tipos son vulnerables a ataques que manipulan el estado interno para provocar condiciones inesperadas.

La mitigación efectiva requiere el diseño de interfaces que **encapsulen el estado interno** y validen todas las modificaciones. Los constructores y métodos de mutación deben verificar que los nuevos valores mantienen los invariantes antes de aplicar cambios. La validación debe ser exhaustiva y cubrir todos los casos límite.

El uso de **tipos opacos** que ocultan la representación interna previene el acceso directo a campos que podrían comprometer los invariantes. Los métodos públicos actúan como puntos de control que garantizan la validez de todas las operaciones.

La implementación de **validación en constructores** asegura que ningún valor inválido pueda existir desde el momento de creación. Esta aproximación establece garantías desde el inicio del ciclo de vida del objeto y simplifica el mantenimiento de invariantes en operaciones posteriores.

Riesgos asociados

- Corromper la seguridad de memoria, permitiendo condiciones de uso después de liberar memoria (use-after-free), desbordamientos de buffer o referencias colgantes.
- Saltarse garantías del sistema de tipos de Rust, lo que puede eliminar protecciones tecnológicas clave contra múltiples familias de vulnerabilidades.
- Introducción de comportamientos indefinidos, con resultados inesperados o ejecución de código malicioso.
- Ataques de escalada de privilegios o ejecución de código arbitrario debido a manipulación incorrecta o invariante de los datos.
- Filtración de información sensible o alteración de datos, al no validar o conservar correctamente las restricciones de tipo.
- Corrupción de datos y fallos lógicos que pueden hacer que aplicaciones críticas fallen en producción.
- Pérdida de portabilidad, ya que el código se puede volver dependiente de comportamientos no garantizados entre compiladores o versiones de Rust.
- Falsos negativos en herramientas de análisis estático: los chequeos del sistema de tipos pueden volverse ineficaces, ocultando vulnerabilidades reales.

Buenas prácticas

- Encapsular los campos y exponer solo constructores que validen invariantes antes de crear la instancia.

```
//-- Estructura con campos públicos sin validación de invariantes
pub struct Edad {
  pub valor: u8,
}
```

```
//-- Estructura con campos privados y constructor que valida rango permitido
pub struct Edad {
  valor: u8,
```

```
}
impl Edad {
  pub fn nueva(valor: u8) -> Option<Self> {
    if valor <= 120 {
      Some(Self { valor })
    } else {
      None
    }
  }
}
```

- Evitar exponer estados intermedios inválidos; usar funciones que mantengan el objeto siempre consistente.

```
//-- Modificación directa que permite saldo negativo temporal
pub struct Cuenta {
  pub saldo: i32,
}
```

```
//-- Métodos que validan operaciones y preservan el invariante de no negatividad
pub struct Cuenta {
  saldo: i32,
}
impl Cuenta {
  pub fn retirar(&mut self, monto: i32) -> Result<(), &'static str> {
    if monto <= self.saldo {
      self.saldo -= monto;
      Ok(())
    } else {
      Err("Saldo insuficiente")
    }
  }
}
```

- Utilizar tipos específicos en lugar de tipos primitivos cuando exista un invariante semántico.

```
//-- Uso de String sin validación de formato
pub struct Usuario {
  pub email: String,
}
```

```
//-- Nuevo tipo que valida formato al construirse
pub struct Email(String);
impl Email {
  pub fn nuevo(valor: String) -> Option<Self> {
    if valor.contains("@") {
```

```
      Some(Self(valor))
    } else {
      None
    }
  }
}
pub struct Usuario {
  email: Email,
}
```

- Evitar el uso de `unsafe` que pueda romper invariantes de tipo; preferir abstracciones seguras.

```
//-- Conversión insegura que puede violar el invariante de inicialización
let valor: u8 = unsafe { std::mem::MaybeUninit::uninit().assume_init() };
```

```
//-- Inicialización explícita que garantiza valor válido
let valor: u8 = 0;
```

- Validar datos externos antes de transformarlos en tipos internos que asumen invariantes.

```
//-- Conversión directa desde entrada externa sin validación
let puerto: u16 = input.parse().unwrap();
```

```
//-- Validación explícita antes de aceptar el valor como válido
let puerto: u16 = match input.parse() {
  Ok(p) if p >= 1024 && p <= 65535 => p,
  _ => return Err("Puerto inválido"),
};
```

2.1.3.2 Estándares metodológicos

- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 3.1**: References.

2.2 Gestión de memoria

2.2.1 [rust.mem.1] - No introducir bloques unsafe sin invariantes de seguridad explícitos, completos y verificables

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

Los bloques `unsafe` permiten realizar operaciones que el compilador no puede verificar automáticamente, como acceso directo a memoria o llamadas a funciones externas. Sin **invariantes de seguridad** documentados y verificables, estos bloques pueden introducir vulnerabilidades graves como corrupción de memoria, accesos fuera de límites o condiciones de carrera. La ausencia de invariantes explícitos dificulta el mantenimiento del código y aumenta el riesgo de errores durante modificaciones posteriores.

Codificación insegura

Uso de un bloque `unsafe` para convertir un vector de bytes en una cadena UTF-8 sin documentar ni validar las condiciones necesarias para que la operación sea segura. La intención es mostrar cómo una conversión aparentemente simple puede introducir comportamiento indefinido si no se explican y verifican las invariantes requeridas.

```
use std::str;

fn main() {
  //-- Se crea un vector de bytes que no es UTF-8 válido
  let bytes = vec![0xff, 0xfe, 0xfd];

  //-- Conversión insegura sin validar invariantes
  let s = unsafe {
    //-- Se asume incorrectamente que los bytes son UTF-8 válido
    str::from_utf8_unchecked(&bytes)
  };

  //-- Uso de la cadena resultante
  println!("Cadena: {}", s);
}
```

Codificación recomendada

Uso de un bloque `unsafe` para convertir un vector de bytes en una cadena UTF-8, documentando de forma explícita las invariantes de seguridad y verificándolas antes de ejecutar la operación insegura. Se muestra cómo reducir el riesgo asegurando que las condiciones requeridas se cumplen de manera comprobable.

```
use std::str;

fn main() {
  //-- Vector de bytes que representa texto UTF-8 válido ("Hola")
  let bytes = vec![72, 111, 108, 97];

  //-- Verificación previa de la invariante: los bytes deben ser UTF-8 válido
  if str::from_utf8(&bytes).is_err() {
    //-- Si no se cumple la invariante, se aborta de forma segura
    panic!("Los bytes no son UTF-8 válido");
  }
  //-- Uso de unsafe con invariante explícita y verificada
  let s = unsafe {
    //-- Invariante: 'bytes' contiene UTF-8 válido verificado arriba
    str::from_utf8_unchecked(&bytes)
  };

  //-- Uso seguro de la cadena resultante
  println!("Cadena: {}", s);
}
```

2.2.1.1 Descripción

Los bloques `unsafe` permiten realizar operaciones que el compilador no puede verificar automáticamente como seguras respecto a la memoria. Estas operaciones incluyen la desreferenciación de punteros sin procesar, el acceso a variables estáticas mutables, la llamada a funciones no seguras y la implementación de traits no seguros.

El uso de código no seguro sin **invariantes de seguridad** documentados introduce vulnerabilidades potenciales en el programa. Los invariantes de seguridad son condiciones que deben mantenerse verdaderas para garantizar que las operaciones no seguras no comprometan la integridad de la memoria ni provoquen comportamientos indefinidos.

Cada bloque `unsafe` debe incluir documentación que especifique claramente qué condiciones de seguridad se asumen, qué operaciones se realizan y por qué estas operaciones son seguras bajo las condiciones establecidas. Esta documentación debe ser **explícita**, cubriendo todos los aspectos relevantes para la seguridad, **completa**, sin omitir condiciones necesarias, y **verificable**, permitiendo que otros desarrolladores puedan validar la corrección de los invariantes.

Los riesgos asociados con bloques `unsafe` mal documentados incluyen corrupción de memoria, accesos fuera de límites, condiciones de carrera y comportamientos indefinidos que pueden ser explotados por atacantes o causar fallos impredecibles del programa.

La mitigación de estos riesgos requiere establecer un proceso de revisión riguroso para todo código no seguro, documentar exhaustivamente los invariantes antes de la implementación, y limitar el alcance de los bloques `unsafe` al mínimo necesario. También resulta efectivo encapsular el código no seguro en interfaces seguras que mantengan los invariantes automáticamente.

Riesgos asociados

- Ejecución de código indefinido ("undefined behavior") por no documentar correctamente las garantías requeridas y asumidas en el bloque `unsafe`.
- Introducción de vulnerabilidades de seguridad, como desbordamientos de memoria, violaciones de memoria o corrupción de datos, debido a la falta de controles explícitos.
- Dificultad para auditar y mantener el código, ya que otros desarrolladores no conocerán las condiciones bajo las cuales el bloque `unsafe` es seguro.
- Posibilidad de que cambios futuros en el código o en dependencias externas rompan las supuestas invariantes, incrementando el riesgo de errores de seguridad.
- El optimizador del compilador podría asumir propiedades incorrectas y producir código inesperado o inseguro.
- Pérdida de las garantías de seguridad que Rust proporciona fuera de los bloques `unsafe`, afectando la confianza en todo el sistema.
- Dificultad para realizar revisiones de código efectivas, ya que no hay una base clara para verificar la seguridad del bloque `unsafe`.
- Incremento de la superficie de ataque del software, facilitando la explotación de vulnerabilidades.

Buenas prácticas

- Documentar explícitamente las invariantes de seguridad antes de cualquier bloque `unsafe` y mantenerlas lo más acotadas posible.

```
//-- Uso de unsafe sin documentar invariantes de validez y alineación
let ptr = 0x12345 as *const i32;
unsafe {
  println!("{}", *ptr);
}
```

```
//-- Uso de unsafe con invariantes explícitas sobre validez y alineación
let value = 10i32;
let ptr = &value as *const i32;
```

```
//-- Invariantes: ptr proviene de una referencia válida, no nula, correctamente
alineada y viva en este alcance
unsafe {
  println!("{}", *ptr);
}
```

- Encapsular el código `unsafe` dentro de funciones seguras que validen las precondiciones antes de invocarlo.

```
//-- Exposición directa de función unsafe sin validación previa
unsafe fn read_ptr(ptr: *const i32) -> i32 {
  *ptr
}
```

```
//-- Encapsulamiento en función segura con validación de puntero no nulo
fn read_ptr(ptr: *const i32) -> Option {
  if ptr.is_null() {
    return None;
  }
  //-- Invariante: ptr no es nulo y se asume correctamente alineado y válido
  Some(unsafe { *ptr })
```

- Reducir el alcance del bloque `unsafe` al mínimo necesario para limitar la superficie de riesgo.

```
//-- Bloque unsafe amplio que incluye lógica no riesgosa
unsafe {
  let value = 5;
  let ptr = &value as *const i32;
  println!("{}", *ptr);
}
```

```
//-- Bloque unsafe limitado únicamente a la desreferenciación
let value = 5;
let ptr = &value as *const i32;
let result = unsafe { *ptr };
println!("{}", result);
```

- Evitar `unsafe` cuando existan alternativas seguras en la biblioteca estándar.

```
//-- Conversión manual de slice usando punteros crudos
let arr = [1, 2, 3];
let ptr = arr.as_ptr();
unsafe {
  let slice = std::slice::from_raw_parts(ptr, 3);
  println!("{:?}", slice);
```

```
}
```

```
//-- Uso directo de referencia segura al slice
let arr = [1, 2, 3];
let slice = &arr[..];
println!("{:?}", slice);
```

- Verificar todas las precondiciones críticas (longitud, alineación, inicialización) antes de operar con memoria sin inicializar o punteros crudos.

```
//-- Lectura de memoria potencialmente no inicializada
let mut buffer: [u8; 4];
unsafe {
  println!("{}", buffer[0]);
}
```

```
//-- Inicialización explícita antes de cualquier acceso
let mut buffer: [u8; 4] = [0; 4];
//-- Invariante: buffer completamente inicializado antes de su uso
println!("{}", buffer[0]);
```

2.2.1.2 Estándares metodológicos

- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rustonomicon 1.1**: How Safe and Unsafe Interact.
- **Rustonomicon 1.2**: What Unsafe Can Do.

2.2.2 [rust.mem.2] - No asumir aliasing válido fuera de las garantías del sistema de tipos

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

El sistema de tipos garantiza que las referencias mutables exclusivas no presenten aliasing, pero estas garantías no se extienden automáticamente a todos los contextos del programa. Asumir incorrectamente que dos punteros no pueden referenciar la misma ubicación de memoria puede generar **comportamiento indefinido** y vulnerabilidades de seguridad cuando las optimizaciones del compilador se basan en suposiciones incorrectas sobre el aliasing.

Codificación insegura

Manipulación de memoria creando múltiples referencias mutables al mismo dato usando código inseguro. Se pretende mostrar cómo romper manualmente las garantías de exclusividad de `&mut` puede generar comportamiento indefinido, incluso si el programa parece funcionar en una ejecución concreta. El objetivo es entender por qué no se debe asumir aliasing válido fuera del sistema de tipos.

```
use std::ptr;

//-- Se demuestra una violación de exclusividad de &mut usando código unsafe
fn main() {
  //-- Variable mutable inicial
  let mut value = 10;

  //-- Se obtiene un puntero crudo al valor
  let ptr1: *mut i32 = &mut value;

  //-- Se crea una segunda referencia mutable al mismo dato
  //-- Esto rompe la garantía de exclusividad de &mut
  let ptr2: *mut i32 = ptr1;

  unsafe {
    //-- Se crean dos referencias mutables simultáneas al mismo dato
    let ref1: &mut i32 = &mut *ptr1;
    let ref2: &mut i32 = &mut *ptr2;

    //-- Escrituras concurrentes sobre el mismo valor
    *ref1 += 5;
    *ref2 += 10;
  }
  //-- El resultado es comportamiento indefinido
  println!("Resultado: {}", value);
}
```

Codificación recomendada

Manipulación segura de un dato mutable respetando la exclusividad garantizada por el sistema de tipos. Se pretende mostrar cómo estructurar el acceso mutable para que nunca existan dos referencias mutables activas al mismo tiempo, delegando en el compilador la validación de aliasing y evitando comportamiento indefinido.

```
fn incrementar(valor: &mut i32) {
  //-- Se modifica el valor usando una única referencia mutable
  *valor += 5;
}
fn main() {
  //-- Variable mutable inicial
```

```
    let mut value = 10;

    //-- Se pasa una única referencia mutable a la función
    incrementar(&mut value);

    //-- Una vez finalizado el préstamo, se puede volver a usar
    value += 10;

    //-- El compilador garantiza que no hubo aliasing inválido
    println!("Resultado: {}", value);
}
```

2.2.2.1 Descripción

El sistema de tipos garantiza que las referencias mutables exclusivas (`&mut`) no presenten aliasing, proporcionando optimizaciones automáticas y prevención de condiciones de carrera. Sin embargo, existen mecanismos que permiten eludir estas garantías, como el código `unsafe`, tipos de interior mutability y punteros raw.

Asumir incorrectamente que múltiples referencias apuntan a ubicaciones de memoria distintas cuando en realidad comparten la misma dirección puede generar **comportamiento indefinido**. Esta situación se produce especialmente al trabajar con punteros raw o al implementar estructuras de datos complejas que requieren múltiples referencias mutables simultáneas.

Los riesgos asociados incluyen corrupción de datos, violaciones de invariantes de tipos, optimizaciones incorrectas del compilador y condiciones de carrera en entornos concurrentes. El compilador puede realizar transformaciones de código basándose en la asunción de no-aliasing, lo que puede resultar en comportamientos inesperados cuando esta premisa es incorrecta.

La mitigación efectiva requiere **validación explícita** de las condiciones de aliasing antes de realizar operaciones que dependan de la exclusividad de acceso. En contextos `unsafe`, es necesario documentar y verificar manualmente las garantías de aliasing. El uso de herramientas de análisis estático y testing exhaustivo ayuda a identificar violaciones potenciales.

Para estructuras de datos que requieren múltiples referencias mutables, se deben emplear patrones de diseño que mantengan las garantías del sistema de tipos, como la división de estructuras o el uso de índices en lugar de referencias directas.

Riesgos asociados

- Condiciones de carrera (data races) debido a accesos concurrentes no controlados a través de alias mutables.
- Comportamiento indefinido por violar la exclusividad de `&mut`, lo que puede

llevar a fallos impredecibles en tiempo de ejecución.

- Corrupción de memoria debido a modificaciones inesperadas a través de múltiples referencias alias.
- Divulgación o fuga de información confidencial al permitir accesos no controlados a datos sensibles mediante alias indebidos.
- Dificultad para detectar errores de seguridad, dado que el compilador asume ciertas garantías sobre la no aliasing que pueden estar siendo violadas.
- Pérdida de integridad en los datos por sobrescritura o lecturas inconsistentes provocadas por referencias solapadas.
- Exposición a vulnerabilidades de denegación de servicio (DoS) si los datos sufren corrupción y el programa falla o entra en bucles erróneos.

Buenas prácticas

- No crear múltiples referencias mutables simultáneas al mismo dato; respetar la exclusividad de `&mut`.

```
//-- creación de múltiples referencias mutables al mismo dato usando punteros
crudos
let mut value = 10;
let r1 = &mut value as *mut i32;
let r2 = &mut value as *mut i32;
unsafe {
  *r1 += 1;
  *r2 += 1;
}
```

```
//-- uso de una única referencia mutable dentro de un alcance controlado
let mut value = 10;
{
  let r = &mut value;
  *r += 1;
}
value += 1;
```

- No asumir que los punteros crudos cumplen reglas de noalias; convertirlos a referencias solo cuando se garantice exclusividad.

```
//-- conversión de puntero crudo a múltiples referencias mutables sin garantizar
exclusividad
fn update(ptr: *mut i32) {
  unsafe {
    let a = &mut *ptr;
    let b = &mut *ptr;
```

```
    *a += 1;
    *b += 1;
  }
}
```

```
//-- validación previa de exclusividad antes de crear una única referencia mutable
fn update(ptr: *mut i32) {
  unsafe {
    let a = &mut *ptr;
    *a += 1;
  }
}
```

- No mezclar referencias inmutables y mutables activas sobre el mismo dato en contextos `unsafe`.

```
//-- coexistencia de referencia inmutable y mutable mediante uso de unsafe
let mut data = 5;
let r1 = &data as *const i32;
let r2 = &mut data as *mut i32;
unsafe {
  println!("{}", *r1);
  *r2 = 20;
}
```

```
//-- separación clara de los alcances de préstamo inmutable y mutable
let mut data = 5;
{
  let r1 = &data;
  println!("{}", r1);
}
{
  let r2 = &mut data;
  *r2 = 20;
}
```

- No exponer referencias mutables internas que puedan generar aliasing externo; encapsular el acceso mediante métodos seguros.

```
//-- devolución directa de referencia mutable interna permitiendo aliasing externo
struct Counter { value: i32 }
impl Counter {
  fn get_mut(&mut self) -> &mut i32 {
    &mut self.value
  }
}
```

```
//-- modificación controlada mediante método que preserva invariantes y
exclusividad
struct Counter { value: i32 }
impl Counter {
  fn increment(&mut self) {
    self.value += 1;
  }
}
```

2.2.2.2 Estándares metodológicos

- **Rust-5.DEREF**: Only smart pointers implement Deref and DerefMut.
- **Rustonomicon 3.2**: Aliasing.
- **Rustonomicon 8.2**: Send and Sync.

2.2.3 [rust.mem.3] - No depender del layout de memoria sin usar repr(C), repr(transparent) o repr(packed) cuando proceda

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

Las estructuras y enumeraciones pueden experimentar cambios en su disposición de memoria entre compilaciones, ya que el compilador optimiza automáticamente el layout para mejorar el rendimiento. Esta reorganización automática genera problemas de **compatibilidad binaria** e **interoperabilidad** cuando el código asume un orden específico de campos en memoria, especialmente en comunicaciones de red, serialización de datos o interfaces con bibliotecas externas.

Codificación insegura

Se define una estructura que se envía como secuencia de bytes a través de la red. La intención es mostrar cómo convertir directamente una estructura a bytes apoyándose en su disposición en memoria, para entender los riesgos de asumir que el compilador mantendrá un orden y alineación específicos sin declararlos explícitamente.

```
use std::mem;

//-- Estructura que representa un mensaje de red
struct Message {
  id: u32,
  flag: u8,
}
fn main() {
  //-- Crear instancia del mensaje
```

```
  let msg = Message { id: 0x11223344, flag: 1 };

  //-- Convertir la estructura directamente a bytes usando su layout en memoria
  let size = mem::size_of::<Message>();
  let ptr = &msg as *const Message as *const u8;

  //-- Crear slice de bytes a partir del puntero crudo
  let bytes = unsafe { std::slice::from_raw_parts(ptr, size) };

  //-- Mostrar los bytes enviados
  println!("Bytes enviados: {:x?}", bytes);
}
```

Codificación recomendada

Se define la misma estructura destinada a enviarse como bytes, pero declarando explícitamente su representación en memoria para garantizar un layout estable y predecible. El objetivo es mostrar cómo fijar el orden y alineación cuando se depende del layout para interoperabilidad o serialización binaria.

```
use std::mem;

//-- Estructura con representación en memoria garantizada estilo C
#[repr(C)]
struct Message {
  id: u32,
  flag: u8,
}
fn main() {
  //-- Crear instancia del mensaje
  let msg = Message { id: 0x11223344, flag: 1 };

  //-- Convertir la estructura a bytes con layout predecible
  let size = mem::size_of::<Message>();
  let ptr = &msg as *const Message as *const u8;

  //-- Crear slice de bytes a partir del puntero crudo
  let bytes = unsafe { std::slice::from_raw_parts(ptr, size) };

  //-- Mostrar los bytes enviados
  println!("Bytes enviados: {:x?}", bytes);
}
```

2.2.3.1 Descripción

El layout de memoria de las estructuras y enumeraciones no está garantizado por defecto. El compilador puede reorganizar, reordenar o añadir padding a los campos para

optimizar el rendimiento o el uso de memoria. Esta flexibilidad permite al compilador generar código más eficiente, pero introduce incertidumbre cuando el código necesita un layout específico.

La dependencia implícita del layout de memoria genera **comportamiento indefinido** cuando se realizan operaciones que asumen un orden o tamaño específico de los campos. Esto incluye la transmutación entre tipos, la serialización binaria, la interoperabilidad con código externo, y el acceso directo a memoria a través de punteros calculados manualmente.

Los riesgos se materializan especialmente en interfaces con bibliotecas externas, protocolos de red, formatos de archivo binarios, y sistemas embebidos donde el layout debe coincidir con especificaciones hardware. Las diferencias entre versiones del compilador, arquitecturas de destino, o configuraciones de optimización pueden alterar el layout y romper el código que depende de él.

La mitigación requiere el uso explícito de atributos de representación cuando se necesita control sobre el layout. `repr(C)` garantiza compatibilidad con el layout de C, `repr(transparent)` asegura que un wrapper tenga el mismo layout que el tipo envuelto, y `repr(packed)` elimina el padding entre campos. Estos atributos proporcionan las garantías necesarias para operaciones que requieren un layout específico.

La verificación del layout puede realizarse mediante pruebas que comprueben tamaños, alineaciones y offsets de campos usando las funciones estándar correspondientes. Esta aproximación detecta cambios inesperados en el layout durante el desarrollo y mantenimiento del código.

Riesgos asociados

- Desbordamiento de memoria o corrupción de datos al acceder a estructuras con layout inesperado.
- Comportamiento indefinido cuando se transmite información entre Rust y código externo (FFI) si la disposición de los campos no está garantizada.
- Fugas de información sensible si partes no inicializadas o no alineadas de la memoria se exponen inintencionadamente.
- Errores en serialización/deserialización de estructuras al intercambiar información binaria con otros sistemas, provocando datos corruptos o inoperantes.
- Dificultad para realizar validación y revisión de código debido a la ambigüedad del layout de estructuras.
- Incompatibilidad y fallos inesperados al utilizar crates o bibliotecas que esperan cierto layout de memoria específico.
- Explotación de vulnerabilidades por parte de atacantes, al aprovechar supuestos

erróneos sobre la disposición interna de las estructuras en memoria.

Buenas prácticas

- Declarar explícitamente la representación con `repr(C)` cuando una estructura sea compartida mediante FFI o con código externo.

```
//-- Estructura usada en FFI sin representación definida
struct Data {
  id: u32,
  flag: u8,
}
```

```
//-- Estructura con layout estable para interoperabilidad C
#[repr(C)]
struct Data {
  id: u32,
  flag: u8,
}
```

- Utilizar `repr(transparent)` en newtypes cuando se requiera garantizar el mismo layout que el tipo interno.

```
//-- Newtype usado en FFI sin garantizar layout equivalente
struct UserId(u32);
```

```
//-- Newtype con layout idéntico al tipo interno
#[repr(transparent)]
struct UserId(u32);
```

- Evitar asumir el orden o padding de los campos para serialización manual; definir representación explícita o usar mecanismos seguros de serialización.

```
//-- Serialización basada en reinterpretación de memoria
struct Packet {
  code: u16,
  value: u32,
}
let bytes = unsafe {
  std::slice::from_raw_parts(
    &(&packet as *const Packet) as *const u8,
    std::mem::size_of::(),
  )
};
```

```
//-- Serialización controlada con layout definido
```

```
#[repr(C)]
struct Packet {
  code: u16,
  value: u32,
}
let bytes = unsafe {
  std::slice::from_raw_parts(
    \(&packet as *const Packet) as *const u8,
    std::mem::size_of::(),
  )
};
```

- Usar `repr(packed)` solo cuando sea estrictamente necesario y gestionar accesos potencialmente desalineados de forma segura.

```
//-- Estructura alineada por defecto asumida como compacta
struct Header {
  version: u8,
  length: u32,
}
```

```
//-- Estructura con layout compacto explícito
#[repr(packed)]
struct Header {
  version: u8,
  length: u32,
}
```

2.2.3.2 Estándares metodológicos

- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rustonomicon 2.1**: repr(Rust).
- **Rustonomicon 2.3**: Other reprs.

2.2.4 [rust.mem.4] - No realizar accesos fuera de límites en arrays, slices o regiones apuntadas por punteros

ID	**rust.mem.4**	**Uso**	**Recomendado**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los accesos fuera de límites en estructuras de datos como arrays, slices y regiones de memoria referenciadas por punteros representan una fuente común de vulnerabilidades de seguridad. Estas operaciones pueden provocar **corrupción de**

memoria, lectura de datos no autorizados o ejecución de código malicioso, comprometiendo la integridad del programa y la seguridad del sistema.

Codificación insegura

Acceso a un elemento de un vector usando un índice calculado dinámicamente sin validar sus límites. Se busca mostrar qué ocurre cuando se intenta leer una posición que no existe y cómo esto impacta en la estabilidad del programa.

```
fn main() {
  //-- Se crea un vector con tres elementos
  let datos = vec![10, 20, 30];

  //-- Índice calculado dinámicamente (simulación de entrada externa)
  let indice: usize = 5;

  //-- Acceso directo sin validar límites
  //-- Esto provocará un panic en tiempo de ejecución
  let valor = datos[indice];

  //-- Esta línea no se ejecutará si ocurre el panic
  println!("Valor obtenido: {}", valor);
}
```

Codificación recomendada

Acceso seguro a un elemento de un vector utilizando métodos que validan los límites antes de acceder a la posición solicitada. Se muestra cómo manejar correctamente el caso en que el índice esté fuera de rango sin provocar una interrupción del programa.

```
fn main() {
  //-- Se crea un vector con tres elementos
  let datos = vec![10, 20, 30];

  //-- Índice calculado dinámicamente (simulación de entrada externa)
  let indice: usize = 5;

  //-- Acceso seguro usando el método get
  //-- Devuelve Option y evita panic por acceso fuera de rango
  match datos.get(indice) {
    Some(valor) => {
      //-- Se procesa el valor solo si existe
      println!("Valor obtenido: {}", valor);
    }
    None => {
      //-- Manejo controlado del error sin detener el programa
      println!("Índice fuera de rango");
    }
```

```
    }
}
```

2.2.4.1 Descripción

Los accesos fuera de límites constituyen una de las vulnerabilidades más frecuentes en sistemas de bajo nivel. Esta regla establece que todas las operaciones de acceso a memoria mediante índices deben verificar que la posición solicitada se encuentre dentro de los límites válidos del contenedor de datos.

El problema surge cuando se intenta acceder a posiciones de memoria que exceden el tamaño asignado a estructuras como arrays, slices o regiones referenciadas por punteros. Estos accesos pueden provocar **corrupción de memoria**, **exposición de información sensible** o **terminación inesperada del programa**.

La verificación de límites debe aplicarse tanto en operaciones de lectura como de escritura. Los índices negativos, cuando se interpretan como valores sin signo de gran magnitud, representan un caso particular de acceso fuera de límites que requiere atención específica.

Las operaciones aritméticas sobre índices introducen complejidad adicional, ya que pueden generar desbordamientos que resulten en índices aparentemente válidos, pero que apunten a posiciones incorrectas. La validación debe considerar tanto el resultado final como los cálculos intermedios.

La mitigación efectiva combina **verificaciones automáticas en tiempo de ejecución** con **análisis estático durante la compilación**. Las verificaciones automáticas detectan violaciones de límites y terminan la ejecución de forma controlada, mientras que el análisis estático puede identificar patrones de acceso potencialmente problemáticos antes de la ejecución.

Los iteradores proporcionan una alternativa que elimina la necesidad de gestión manual de índices, reduciendo significativamente la superficie de ataque. Cuando el acceso directo por índice resulta necesario, las funciones que retornan tipos `Option` permiten manejar casos de límites excedidos de forma explícita y segura.

Riesgos asociados

- Lectura de datos sensibles fuera del rango previsto, lo que puede provocar exposición de información confidencial.
- Corrupción de memoria, pudiendo causar comportamientos inesperados o inestables en el programa.
- Producción de errores en ejecución (panics), que pueden causar la terminación abrupta de la aplicación.
- Ejecución de código con resultados incorrectos debido a la utilización de datos

inválidos o no inicializados.

- Vulnerabilidades de seguridad, como ataques de desbordamiento de buffer, que pueden ser aprovechadas por actores maliciosos para tomar control del programa.
- Condiciones de denegación de servicio (DoS) si los accesos fuera de límites generan panics o fallos continuos.
- Difícil detección y depuración de errores, especialmente en uso de código inseguro (`unsafe`), afectando la mantenibilidad.

Buenas prácticas

- Validar explícitamente los límites antes de acceder a un índice en arrays o slices.

```
//-- Acceso directo sin validar el índice
let arr = [10, 20, 30];
let i = 5;
let value = arr[i];
```

```
//-- Validación previa del índice antes de acceder
let arr = [10, 20, 30];
let i = 5;
if i < arr.len() {
  let value = arr[i];
}
```

- Preferir métodos seguros como `get()` en lugar de indexación directa cuando el índice no está garantizado.

```
//-- Indexación directa que puede provocar panic
let data = vec![1, 2, 3];
let value = data[10];
```

```
//-- Uso de get() para manejar acceso fuera de rango
let data = vec![1, 2, 3];
if let Some(value) = data.get(10) {
  println!("{}", value);
}
```

- En código `unsafe`, comprobar manualmente los límites antes de realizar aritmética de punteros.

```
//-- Aritmética de puntero sin validación de límites
let arr = [5, 6, 7];
let ptr = arr.as_ptr();
```

```
unsafe {
  let value = *ptr.add(5);
}
```

```
//-- Validación manual antes de desreferenciar el puntero
let arr = [5, 6, 7];
let ptr = arr.as_ptr();
let index = 1;
if index < arr.len() {
  unsafe {
    let value = *ptr.add(index);
  }
}
```

- Evitar convertir slices o buffers sin validar previamente su tamaño esperado.

```
//-- Asumir tamaño mínimo sin verificación previa
let buffer: [u8; 2] = [1, 2];
let slice = &buffer[0..4];
```

```
//-- Comprobar longitud antes de crear el slice
let buffer: [u8; 2] = [1, 2];
if buffer.len() >= 4 {
  let slice = &buffer[0..4];
}
```

2.2.4.2 Estándares metodológicos

- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 5.1**: Checked.
- **Rustonomicon 5.3**: Unchecked.

2.2.5 [rust.mem.5] - No permitir uso-after-free, doble liberación ni referencias colgantes

ID	**rust.mem.5**	**Uso**	**Recomendado**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los errores de gestión de memoria como **uso después de liberación**, **doble liberación** y **referencias colgantes** representan vulnerabilidades críticas que pueden provocar corrupción de memoria, comportamiento indefinido y fallos de seguridad. Estos problemas surgen cuando el código accede a memoria ya liberada, libera el mismo

bloque de memoria múltiples veces, o mantiene punteros que apuntan a ubicaciones de memoria inválidas.

Codificación insegura

Gestión manual de memoria utilizando punteros crudos para mostrar cómo una liberación explícita seguida de un acceso posterior puede provocar una referencia colgante. Se pretende observar cómo el uso de bloques `unsafe` puede romper las garantías de seguridad del compilador y generar un escenario de uso- después- de- liberación.

```
fn main() {
  //-- Reserva memoria en el heap usando Box
  let b = Box::new(10);

  //-- Convierte Box en puntero crudo, transfiriendo la responsabilidad de
liberación
  let ptr = Box::into_raw(b);

  //-- Libera manualmente la memoria
  unsafe {
    Box::from_raw(ptr);
  }
  //-- Accede al puntero después de liberar la memoria (uso-after-free)
  unsafe {
    println!("Valor: {}", *ptr);
  }
}
```

Codificación recomendada

Uso correcto del modelo de propiedad y préstamos de Rust para garantizar que la memoria se libera automáticamente sin accesos posteriores inválidos. Se busca demostrar cómo evitar punteros crudos y delegar en el compilador la prevención de referencias colgantes y dobles liberaciones.

```
fn main() {
  //-- Reserva memoria en el heap usando Box
  let b = Box::new(10);

  //-- Se crea un préstamo inmutable dentro de un alcance controlado
  {
    let reference = &*b;
    //-- Uso seguro mientras el propietario sigue siendo válido
    println!("Valor: {}", reference);
  }
  //-- La memoria se libera automáticamente al final del alcance de b
```

```
}
```

2.2.5.1 Descripción

Esta regla prohíbe el acceso a memoria que ha sido previamente liberada, la liberación múltiple del mismo bloque de memoria y el mantenimiento de referencias que apuntan a ubicaciones de memoria inválidas. Estos problemas constituyen vulnerabilidades críticas de gestión de memoria que pueden comprometer tanto la estabilidad como la seguridad de las aplicaciones.

El uso-after-free ocurre cuando el código intenta acceder a memoria después de que esta ha sido desasignada. La doble liberación se produce al intentar liberar el mismo bloque de memoria más de una vez. Las referencias colgantes aparecen cuando una referencia continúa apuntando a una ubicación de memoria que ya no es válida o ha sido reasignada para otros propósitos.

Estas condiciones pueden provocar **corrupción de memoria**, comportamiento indefinido del programa y fallos inesperados del sistema. Desde una perspectiva de seguridad, representan vectores de ataque que pueden ser explotados para ejecutar código arbitrario o acceder a información sensible.

El sistema de ownership y borrowing proporciona verificaciones en tiempo de compilación que previenen automáticamente estos problemas. El compilador rastrea la vida útil de las referencias y garantiza que no puedan existir referencias a memoria inválida. Los tipos inteligentes como `Rc`, `Arc` y `Box` ofrecen mecanismos adicionales de gestión segura de memoria.

La mitigación efectiva requiere comprender los principios de ownership, utilizar los mecanismos de borrowing checker y emplear los tipos de datos apropiados para cada contexto. En situaciones que requieren código `unsafe`, es necesario implementar verificaciones manuales rigurosas para mantener las garantías de seguridad de memoria.

Riesgos asociados

- Ejecutar operaciones sobre memoria liberada, lo que puede derivar en bloqueos inesperados (crash) de la aplicación.
- Exposición a vulnerabilidades de corrupción de memoria, que pueden ser explotadas para ejecución arbitraria de código.
- Elevación de privilegios por parte de un atacante al manipular estructuras de memoria tras su liberación.
- Filtración o revelación de datos sensibles almacenados en ubicaciones de memoria reutilizadas.
- Comportamiento indefinido del programa, dificultando el mantenimiento y

depuración.

- Incremento del riesgo de condicionar estados de carrera (race conditions) y condiciones de carrera en hilos concurrentes.
- Inestabilidad general de la aplicación, con afectación potencial a la integridad y disponibilidad del servicio.
- Vulneración de garantías de seguridad y fiabilidad que ofrece Rust, perdiendo uno de sus mayores beneficios.

Buenas prácticas

- Transferir la propiedad correctamente usando el sistema de ownership en lugar de duplicar o reutilizar referencias después de liberar memoria.

```
//-- Se usa memoria después de liberar manualmente
let b = Box::new(10);
let ptr = Box::into_raw(b);
unsafe { Box::from_raw(ptr); }
unsafe { println!("{}", *ptr); }

//-- Se mantiene la propiedad y se evita acceso tras liberación
let b = Box::new(10);
println!("{}", b);
```

- Evitar dobles liberaciones asegurando que cada recurso tenga un único propietario y no reconstruyendo múltiples veces desde el mismo puntero crudo.

```
//-- Se reconstruye dos veces el mismo puntero
let b = Box::new(20);
let ptr = Box::into_raw(b);
unsafe { Box::from_raw(ptr); }
unsafe { Box::from_raw(ptr); }

//-- Se reconstruye una sola vez y se respeta ownership
let b = Box::new(20);
let ptr = Box::into_raw(b);
let b = unsafe { Box::from_raw(ptr) };
println!("{}", b);
```

- No retornar referencias a datos locales; utilizar ownership o tipos como `String` en lugar de referencias a variables que saldrán de alcance.

```
//-- Se retorna referencia a variable local
fn obtener() -> &str {
  let s = String::from("hola");
  &s
}
```

```
//-- Se retorna el valor transfiriendo ownership
fn obtener() -> String {
  let s = String::from("hola");
  s
}
```

2.2.5.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 6.2**: Destructors.

2.2.6 [rust.mem.6] - No leer memoria no inicializada ni parcialmente inicializada

ID	rust.mem.6	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La lectura de memoria no inicializada o parcialmente inicializada constituye una fuente común de **comportamiento indefinido** que puede provocar fallos de seguridad y errores impredecibles en tiempo de ejecución. Este problema surge cuando se accede a variables, estructuras o arrays antes de asignarles valores válidos, lo que resulta en la interpretación de datos residuales presentes en memoria.

Codificación insegura

Uso de `MaybeUninit` para trabajar con memoria sin inicializar. Se muestra cómo se puede crear una variable sin valor inicial y acceder a ella directamente, lo que permite observar el riesgo de leer memoria no inicializada y el impacto que puede tener en la seguridad y estabilidad del programa.

```
use std::mem::MaybeUninit;

fn main() {
  //-- Se declara una variable sin inicializar
  let x: MaybeUninit = MaybeUninit::uninit();
  //-- Se asume incorrectamente que la variable ya está inicializada
  let valor: u32 = unsafe { x.assume_init() };
  //-- Se imprime el valor leído desde memoria no inicializada
  println!("Valor leído: {}", valor);
}
```

Codificación recomendada

Uso de `MaybeUninit` asegurando que la memoria se inicializa completamente antes de ser leída. Se inicializa explícitamente el valor y solo después se convierte en un valor válido, evitando el acceso a memoria no inicializada.

```
use std::mem::MaybeUninit;

fn main() {
  //-- Se declara una variable sin inicializar
  let mut x: MaybeUninit = MaybeUninit::uninit();
  //-- Se escribe un valor válido en la memoria antes de usarla
  x.write(42);
  //-- Ahora es seguro asumir que la variable está inicializada
  let valor: u32 = unsafe { x.assume_init() };
  //-- Se imprime el valor correctamente inicializado
  println!("Valor leído: {}", valor);
}
```

2.2.6.1 Descripción

La lectura de memoria no inicializada o parcialmente inicializada constituye una fuente común de vulnerabilidades y comportamientos impredecibles en aplicaciones. Esta regla establece que no se debe acceder a valores almacenados en ubicaciones de memoria que no han sido explícitamente inicializadas con datos válidos antes de su uso.

El problema surge cuando se declaran variables, estructuras o arrays sin asignarles valores iniciales, y posteriormente se intenta leer su contenido. En estos casos, la memoria puede contener datos residuales de operaciones anteriores del programa o del sistema operativo, generando resultados no deterministas que varían entre ejecuciones.

Los riesgos asociados incluyen la exposición de información sensible que pudiera haber permanecido en memoria de sesiones previas, la generación de comportamientos erráticos en la lógica del programa, y la creación de vulnerabilidades de seguridad explotables por atacantes. Estos problemas se agravan en entornos donde la memoria no se limpia automáticamente entre asignaciones.

La mitigación efectiva requiere establecer valores iniciales explícitos para todas las variables antes de su primer uso. Esto incluye la inicialización completa de estructuras complejas, arrays y tipos de datos compuestos. Las herramientas de análisis estático pueden detectar muchos casos de uso de variables no inicializadas durante el proceso de compilación.

El sistema de tipos y el compilador proporcionan mecanismos integrados para prevenir el acceso a memoria no inicializada mediante verificaciones en tiempo de compilación. Estas verificaciones identifican patrones de código donde se intenta leer variables antes

de asignarles valores, bloqueando la compilación hasta que se resuelvan estos problemas.

La aplicación consistente de esta regla mejora la **predictibilidad del comportamiento** del programa, reduce la superficie de ataque disponible para explotación, y facilita el proceso de depuración al eliminar una fuente común de errores intermitentes y difíciles de reproducir.

Riesgos asociados

- Exposición de información sensible: la memoria no inicializada puede contener datos residuales de otras variables, estructuras o procesos, resultando en la posible filtración de datos confidenciales.
- Comportamiento indefinido: leer memoria no inicializada puede llevar a resultados impredecibles, afectando la lógica y el flujo del programa.
- Fallos de seguridad explotables: un atacante podría aprovechar datos aleatorios leídos de memoria para alterar el estado interno del programa, provocar errores o encontrar vectores de ataque.
- Dificultad en el diagnóstico de errores: los bugs relacionados con memoria no inicializada suelen ser difíciles de reproducir y depurar debido a su naturaleza aleatoria.
- Posible ejecución de código malicioso: en casos extremos, el uso de datos sin inicializar puede derivar en vulnerabilidades aprovechables para la ejecución de código arbitrario.
- Corrupción de datos: el uso de valores aleatorios provenientes de memoria no inicializada puede causar la alteración o destrucción de datos legítimos durante la ejecución del programa.

Buenas prácticas

- Inicializar siempre las variables antes de su uso, evitando el uso de `MaybeUninit` sin una inicialización completa garantizada.

```
//-- Uso de memoria potencialmente no inicializada con MaybeUninit
let x: std::mem::MaybeUninit = std::mem::MaybeUninit::uninit();
let y = unsafe { x.assume_init() };
println!("{}", y);
```

```
//-- Inicialización explícita antes de acceder al valor
let x: std::mem::MaybeUninit = std::mem::MaybeUninit::new(42);
let y = unsafe { x.assume_init() };
println!("{}", y);
```

- Evitar la creación de estructuras parcialmente inicializadas; utilizar constructores que garanticen la inicialización completa.

```
struct Config {
  host: String,
  port: u16,
}
```

```
//-- Inicialización parcial usando valores por defecto inseguros
let mut cfg: Config = unsafe { std::mem::zeroed() };
cfg.host = "localhost".to_string();

//-- Constructor que asegura inicialización completa
let cfg = Config {
  host: "localhost".to_string(),
  port: 8080,
};
```

- No usar bloques `unsafe` para eludir las comprobaciones del compilador relacionadas con la inicialización.

```
//-- Acceso inseguro a elemento potencialmente no inicializado
let mut data: [std::mem::MaybeUninit; 4] = unsafe {
std::mem::MaybeUninit::uninit().assume_init() };
let value = unsafe { data[0].assume_init() };
println!("{}", value);
```

```
//-- Inicialización segura mediante iteración controlada
let mut data: [u8; 4] = [0; 4];
for i in 0..data.len() {
  data[i] = i as u8;
}
println!("{}", data[0]);
```

2.2.6.2 Estándares metodológicos

- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 5.1**: Checked.
- **Rustonomicon 5.3**: Unchecked.

2.3 Gestión de recursos

2.3.1 [rust.res.1] - No depender de orden implícito de drop para mantener invariantes críticos

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

El orden de destrucción de variables en un mismo ámbito no está garantizado por el compilador, lo que puede generar comportamientos impredecibles cuando múltiples recursos interdependientes requieren una secuencia específica de liberación. Esta indeterminación compromete la **consistencia del programa** y puede provocar estados inválidos o corrupción de datos cuando los destructores dependen unos de otros para mantener invariantes del sistema.

Codificación insegura

Se muestra una estructura que mantiene un invariante crítico mediante dos recursos relacionados. El objetivo es que uno de ellos solo se libere cuando el otro siga activo. El aprendizaje clave es cómo depender del orden implícito de `drop` puede romper ese invariante y provocar fallos en tiempo de ejecución.

```
use std::sync::atomic::{AtomicBool, Ordering};

static ACTIVE: AtomicBool = AtomicBool::new(false);

struct Guard;

impl Guard {
  fn new() -> Self {
    //-- Activa el recurso crítico
    ACTIVE.store(true, Ordering::SeqCst);
    Guard
  }
}
impl Drop for Guard {
  fn drop(&mut self) {
    //-- Marca el recurso como inactivo al liberarse
    ACTIVE.store(false, Ordering::SeqCst);
  }
}
struct Resource;

impl Resource {
  fn new() -> Self {
    Resource
```

```
  }
}
impl Drop for Resource {
  fn drop(&mut self) {
    //-- Verifica que el recurso crítico siga activo
    if !ACTIVE.load(Ordering::SeqCst) {
      panic!("Invariante roto: Resource liberado con ACTIVE=false");
    }
  }
}
struct App {
  //-- Se asume incorrectamente que Resource se liberará antes que Guard
  guard: Guard,
  resource: Resource,
}
fn main() {
  //-- Se crean ambos recursos
  let _app = App {
    guard: Guard::new(),
    resource: Resource::new(),
  };
  //-- Al salir de main, el orden implícito de drop rompe el invariante
}
```

Codificación recomendada

Se presenta la misma lógica de recursos relacionados, pero controlando explícitamente el orden de liberación para garantizar el invariante. El aprendizaje clave es evitar depender del orden implícito de `drop` y definir de forma explícita cómo y cuándo se liberan los recursos críticos.

```
use std::sync::atomic::{AtomicBool, Ordering};

static ACTIVE: AtomicBool = AtomicBool::new(false);

struct Guard;

impl Guard {
  fn new() -> Self {
    //-- Activa el recurso crítico
    ACTIVE.store(true, Ordering::SeqCst);
    Guard
  }
}
impl Drop for Guard {
  fn drop(&mut self) {
    //-- Marca el recurso como inactivo al liberarse
    ACTIVE.store(false, Ordering::SeqCst);
```

```
  }
}
struct Resource;

impl Resource {
  fn new() -> Self {
    Resource
  }
}
impl Drop for Resource {
  fn drop(&mut self) {
    //-- Verifica que el recurso crítico siga activo
    if !ACTIVE.load(Ordering::SeqCst) {
      panic!("Invariante roto: Resource liberado con ACTIVE=false");
    }
  }
}
struct App {
  guard: Option,
  resource: Option,
}
impl App {
  fn new() -> Self {
    App {
      guard: Some(Guard::new()),
      resource: Some(Resource::new()),
    }
  }
}
impl Drop for App {
  fn drop(&mut self) {
    //-- Se libera explícitamente Resource antes que Guard
    if let Some(resource) = self.resource.take() {
      drop(resource);
    }
    //-- Luego se libera Guard manteniendo el invariante
    if let Some(guard) = self.guard.take() {
      drop(guard);
    }
  }
}
fn main() {
  //-- Se crean ambos recursos bajo control explícito
  let _app = App::new();
  //-- Al salir de main, Drop de App controla el orden correcto
}
```

2.3.1.1 Descripción

El sistema de gestión automática de memoria ejecuta destructores siguiendo reglas de orden específicas, pero este comportamiento no debe utilizarse para mantener invariantes de seguridad o corrección del programa. La dependencia del orden de liberación automática de recursos introduce fragilidad en el código y puede generar comportamientos indefinidos cuando las suposiciones sobre la secuencia de destrucción no se cumplen.

El compilador determina el orden de `Drop` basándose en el alcance léxico y las reglas de propiedad, pero estos mecanismos están diseñados para la gestión de memoria, no para garantizar la ejecución ordenada de lógica crítica. Los cambios en la estructura del código, las optimizaciones del compilador o las modificaciones en las dependencias pueden alterar el orden esperado sin generar advertencias explícitas.

Esta dependencia implícita representa un riesgo particular en sistemas concurrentes, donde el orden de liberación puede afectar la sincronización entre hilos, y en aplicaciones que manejan recursos externos como archivos, conexiones de red o dispositivos de hardware. La falta de control explícito sobre la secuencia de limpieza puede provocar condiciones de carrera, corrupción de datos o estados inconsistentes del sistema.

La mitigación de estos riesgos requiere el uso de mecanismos explícitos de control de orden. Las implementaciones de `Drop` deben limitarse a la liberación de recursos específicos del tipo, mientras que la lógica de coordinación debe manejarse mediante llamadas explícitas a métodos de limpieza, patrones de sincronización apropiados o estructuras de datos que garanticen el orden requerido independientemente del sistema automático de gestión de memoria.

El diseño robusto separa claramente la responsabilidad de liberación de memoria de la lógica de negocio que requiere secuencias específicas de ejecución. Esta separación mejora la predictibilidad del comportamiento del programa y reduce la superficie de ataque para errores relacionados con el orden de operaciones.

Riesgos asociados

- Pérdida de invariantes críticos al liberarse los recursos en un orden inesperado.
- Condiciones de carrera en aplicaciones concurrentes debido a la destrucción fuera de orden.
- Vulnerabilidad a ataques de seguridad donde un recurso es accesible después de liberar un invariante dependiente.
- Fugas de memoria o recursos (por ejemplo, archivos o sockets no cerrados) si el orden de `Drop` interrumpe la limpieza adecuada.
- Comportamiento indefinido o panics en tiempo de ejecución cuando se accede a un objeto después de que su dependencia fue destruida.

- Dificultad para auditar y mantener el código, lo que puede llevar a errores sutiles de seguridad o lógica.
- Escalada de privilegios o corrupción de datos por uso inseguro de recursos ya liberados.
- Inconsistencias en el estado de la aplicación, haciendo que ciertos controles o validaciones se salten inadvertidamente.

Buenas prácticas

- Encapsular los recursos críticos en estructuras que implementen `Drop` explícitamente para garantizar el orden y las invariantes.

```
//-- Recursos separados cuyo orden de liberación afecta una invariante crítica
struct FileHandle;
struct Lock;
struct Context {
  file: FileHandle,
  lock: Lock,
}
```

```
//-- Encapsulamiento que controla explícitamente la liberación y mantiene la
invariante
struct SafeContext {
  file: Option,
  lock: Option,
}
impl Drop for SafeContext {
  fn drop(&mut self) {
    self.file.take();
    self.lock.take();
  }
}
```

- No confiar en el orden implícito de los campos dentro de una estructura para lógica de seguridad.

```
//-- Dependencia implícita del orden de declaración de campos
struct Session {
  auth_token: String,
  connection: Connection,
}
```

```
//-- Método explícito para cerrar recursos en orden definido
impl Session {
  fn close(mut self) {
    self.connection.shutdown();
```

```
    self.auth_token.clear();
  }
}
```

- Evitar depender del `Drop` implícito en presencia de panics; utilizar mecanismos explícitos de manejo y liberación controlada.

```
//-- Liberación implícita que puede no preservar el estado consistente ante panic
fn process(resource: CriticalResource) {
  resource.update_state();
  panic!("error inesperado");
}
```

```
//-- Manejo explícito que asegura restauración de estado antes de finalizar
fn process(mut resource: CriticalResource) {
  if let Err(_) = resource.update_state_checked() {
    resource.rollback();
    return;
  }
}
```

- Usar patrones RAII personalizados para garantizar que las invariantes críticas se validen al liberar recursos.

```
//-- Recurso sin validación de invariante al liberarse
struct Transaction {
  committed: bool,
}
```

```
//-- Validación explícita de invariante en Drop
impl Drop for Transaction {
  fn drop(&mut self) {
    if !self.committed {
      self.rollback();
    }
  }
}
```

2.3.1.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 3.9**: Drop Check.

2.3.2 [rust.res.2] - Liberar recursos externos de forma determinista mediante RAII

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

Los recursos externos como archivos, conexiones de red o memoria asignada dinámicamente requieren liberación explícita para evitar **memory leaks** y agotamiento de recursos del sistema. El patrón **RAII** (Resource Acquisition Is Initialization) garantiza que los recursos se liberen automáticamente cuando salen de ámbito, vinculando la gestión de recursos al ciclo de vida de los objetos.

Codificación insegura

Gestión manual de un recurso externo simulado donde se requiere llamar explícitamente a un método de liberación. Se busca mostrar cómo un olvido o una ruta de error puede dejar el recurso sin liberar, generando fugas o estados inconsistentes.

```
use std::process;

//-- Definición de un recurso externo simulado que requiere cierre manual
struct ExternalResource {
  name: String,
  closed: bool,
}
impl ExternalResource {
  //-- Apertura del recurso
  fn open(name: &str) -> Self {
    println!("Abriendo recurso: {}", name);
    ExternalResource {
      name: name.to_string(),
      closed: false,
    }
  }
  //-- Liberación manual del recurso
  fn close(&mut self) {
    if !self.closed {
      println!("Cerrando recurso: {}", self.name);
      self.closed = true;
    }
  }
}
fn main() {
  //-- Apertura del recurso
  let mut resource = ExternalResource::open("conexion_db");
```

```
  //-- Simulación de un error que termina el proceso antes de liberar el recurso
  let error = true;
  if error {
    println!("Ocurrió un error crítico");
    //-- El recurso no se libera porque no se llama a close()
    process::exit(1);
  }
  //-- Esta línea no se ejecuta si hay error
  resource.close();
}
```

Codificación recomendada

Gestión del mismo recurso externo utilizando RAII mediante la implementación del trait `Drop`. La liberación se realiza de forma automática y determinista cuando el objeto sale de ámbito, incluso ante errores controlados, reduciendo el riesgo de fugas.

```
use std::fmt;

//-- Definición de un recurso externo que se liberará automáticamente
struct ExternalResource {
  name: String,
}
//-- Implementación de lógica de apertura
impl ExternalResource {
  fn open(name: &str) -> Self {
    println!("Abriendo recurso: {}", name);
    ExternalResource {
      name: name.to_string(),
    }
  }
}
//-- Implementación de RAII mediante Drop para liberar el recurso automáticamente
impl Drop for ExternalResource {
  fn drop(&mut self) {
    println!("Liberando recurso automáticamente: {}", self.name);
  }
}
//-- Implementación opcional para mostrar el recurso
impl fmt::Display for ExternalResource {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    write!(f, "{}", self.name)
  }
}
fn main() {
  //-- El recurso se crea dentro de un bloque
  {
    let resource = ExternalResource::open("conexion_db");
```

```
    println!("Usando recurso: {}", resource);
    //-- Al salir de este bloque, Drop se ejecuta automáticamente
  }
  //-- En este punto el recurso ya fue liberado de forma determinista
  println!("Fin del programa");
}
```

2.3.2.1 Descripción

El patrón RAII vincula la vida útil de los recursos externos al ámbito léxico de los objetos que los encapsulan. Este enfoque garantiza que la adquisición del recurso ocurre durante la construcción del objeto y su liberación durante la destrucción, eliminando la dependencia de llamadas manuales de limpieza.

La implementación del trait `Drop` permite definir el comportamiento de liberación que se ejecuta automáticamente cuando el objeto sale de ámbito. Este mecanismo proporciona **liberación determinista**, donde el momento exacto de la limpieza está determinado por las reglas de ámbito del lenguaje, no por la actividad del recolector de basura o decisiones del programador.

Los tipos que encapsulan recursos externos deben implementar semánticas de movimiento apropiadas para transferir la propiedad del recurso sin duplicación. La combinación de ownership, borrowing y el sistema de tipos previene el acceso a recursos ya liberados y garantiza que cada recurso tenga un único propietario responsable de su liberación.

La mitigación efectiva requiere encapsular todos los recursos externos en tipos que implementen `Drop` y evitar el acceso directo a handles o punteros del sistema. Los wrappers RAII deben diseñarse para que la liberación sea idempotente y segura ante fallos, manejando apropiadamente los casos donde la liberación del recurso puede fallar.

Riesgos asociados

- Fugas de memoria por no liberar recursos al dejar de utilizarlos.
- Exposición a ataques de denegación de servicio (DoS) por agotamiento de recursos del sistema.
- Bloqueo de archivos, sockets u otros handles externos por no cerrarlos oportunamente.
- Condiciones de competencia o deadlocks si los recursos no se liberan en el orden esperado.
- Dificultad para depurar errores debido a la gestión no determinista de recursos.
- Incremento de la superficie de ataque ante fallos inesperados o caídas del

programa.

- Alteración de la integridad de datos si no se asegura el flush o commit de recursos como archivos o bases de datos.
- Comportamientos inconsistentes en entornos multihilo, donde los recursos puedan permanecer ocupados indefinidamente.

Buenas prácticas

- Encapsular recursos externos en estructuras que implementen `Drop` para garantizar su liberación automática al salir del ámbito.

```
//-- Recurso externo sin liberación determinista
struct Conexion {
  handle: i32,
}
fn usar() {
  let c = Conexion { handle: 10 };
  // uso del recurso
  // no se libera explícitamente
}
```

```
//-- Recurso encapsulado con liberación automática en Drop
struct ConexionSegura {
  handle: i32,
}
impl Drop for ConexionSegura {
  fn drop(&mut self) {
    cerrar(self.handle);
  }
}
fn usar() {
  let c = ConexionSegura { handle: 10 };
  // uso del recurso
}
```

- Evitar la liberación manual dispersa y preferir wrappers RAII para prevenir fugas ante errores o retornos anticipados.

```
//-- Liberación manual dependiente del flujo de ejecución
fn procesar(handle: i32) {
  if validar(handle).is_err() {
    return;
  }
  // uso del recurso
  cerrar(handle);
}
```

```
//-- Wrapper RAII que libera el recurso incluso con retornos anticipados
struct Recurso {
  handle: i32,
}
impl Drop for Recurso {
  fn drop(&mut self) {
    cerrar(self.handle);
  }
}
fn procesar(handle: i32) {
  let r = Recurso { handle };
  if validar(r.handle).is_err() {
    return;
  }
  // uso del recurso
}
```

- Evitar el uso de `std::mem::forget` u operaciones que impidan la ejecución de `Drop` salvo que esté plenamente justificado y controlado.

```
//-- Se evita la ejecución de Drop provocando fuga del recurso
let r = Recurso { handle: 20 };
std::mem::forget(r);
```

```
//-- El recurso sale de ámbito y se ejecuta Drop automáticamente
let r = Recurso { handle: 20 };
// uso del recurso
```

- Limitar el alcance (scope) de los recursos para reducir el tiempo de exposición y asegurar su liberación temprana.

```
//-- Recurso con alcance mayor al necesario
let r = Recurso { handle: 30 };
// otras operaciones no relacionadas
// uso tardío del recurso
```

```
//-- Recurso confinado a un bloque para liberación temprana
{
  let r = Recurso { handle: 30 };
  // uso del recurso
}
// otras operaciones sin mantener el recurso abierto
```

2.3.2.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.

- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.

2.3.3 [rust.res.3] - No filtrar recursos en rutas de error, panic o cancelación

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

Los recursos del sistema como archivos, conexiones de red o memoria asignada dinámicamente requieren liberación explícita tras su uso. Las rutas de ejecución excepcionales, incluyendo condiciones de `panic`, manejo de errores y cancelaciones de operaciones, pueden interrumpir el flujo normal del programa antes de que se ejecuten las instrucciones de limpieza. Esta situación genera **fugas de recursos** que degradan el rendimiento del sistema y pueden provocar agotamiento de recursos disponibles.

Codificación insegura

Gestión manual de un recurso simulado que requiere liberación explícita. Se reserva memoria dinámica y, ante una condición de error, se fuerza una salida prematura que impide ejecutar la liberación, permitiendo observar cómo una mala gestión en rutas de error puede provocar fuga de recursos.

```
use std::mem;

struct Recurso {
  data: Box,
}
impl Recurso {
  fn new(contenido: &str) -> Self {
    //-- Reserva memoria dinámica para simular un recurso
    Recurso {
      data: Box::new(contenido.to_string()),
    }
  }
}
impl Drop for Recurso {
  fn drop(&mut self) {
    //-- Liberación del recurso
    println!("Liberando recurso con valor: {}", self.data);
  }
}
fn procesar_error() {
  //-- Se crea el recurso
  let recurso = Recurso::new("dato sensible");
  //-- Simulación de condición de error
  let error = true;
  if error {
```

```
    //-- Se evita la liberación automática del recurso
    mem::forget(recurso);
    println!("Error detectado, recurso no liberado");
    return;
  }
}
fn main() {
  procesar_error();
  println!("Fin del programa");
}
```

Codificación recomendada

Gestión segura del mismo recurso utilizando RAII y el operador `?` para propagar errores. La liberación queda garantizada automáticamente incluso cuando se produce un error, evitando fugas en rutas de fallo o cancelación.

```
struct Recurso {
  data: Box,
}
impl Recurso {
  fn new(contenido: &str) -> Self {
    //-- Reserva memoria dinámica para simular un recurso
    Recurso {
      data: Box::new(contenido.to_string()),
    }
  }
}
impl Drop for Recurso {
  fn drop(&mut self) {
    //-- Liberación automática del recurso al salir del ámbito
    println!("Liberando recurso con valor: {}", self.data);
  }
}
(), String> {
  //-- Se crea el recurso
  let _recurso = Recurso::new("dato sensible");
  //-- Simulación de condición de error
  let error = true;
  if error {
    //-- Se propaga el error sin impedir la liberación automática
    return Err("Error detectado".to_string());
  }
  Ok(())
}
fn main() {
  match procesar() {
    Ok(_) => println!("Proceso completado correctamente"),
```

```
    Err(e) => println!("Fallo: {}", e),
  }
}
```

2.3.3.1 Descripción

Los recursos del sistema requieren liberación explícita para evitar agotamiento de memoria, descriptores de archivo o conexiones de red. Las rutas de ejecución alternativas como manejo de errores, situaciones de pánico o cancelación de operaciones pueden omitir inadvertidamente estas operaciones de limpieza.

El problema surge cuando el código adquiere recursos durante la ejecución normal, pero no garantiza su liberación en todos los posibles caminos de salida. Las funciones que terminan abruptamente mediante `panic!`, propagación de errores con `?` o retornos tempranos pueden dejar recursos sin liberar.

Esta situación genera **fugas de recursos** que degradan progresivamente el rendimiento del sistema. Los descriptores de archivo no cerrados pueden agotar los límites del sistema operativo, mientras que la memoria no liberada reduce la disponibilidad para otras operaciones. En aplicaciones de larga duración, estas fugas acumulativas pueden provocar fallos del sistema.

La mitigación efectiva requiere estructurar el código para que la liberación de recursos ocurra automáticamente independientemente del camino de salida. Los mecanismos de gestión automática de recursos garantizan que las operaciones de limpieza se ejecuten incluso cuando las funciones terminan inesperadamente.

El patrón RAII (Resource Acquisition Is Initialization) proporciona la base para esta gestión automática. Los tipos que implementan el trait `Drop` ejecutan código de limpieza cuando salen del ámbito, asegurando la liberación de recursos sin intervención manual del programador.

Riesgos asociados

- Divulgación inadvertida de información sensible (por ejemplo, claves, tokens, configuraciones internas).
- Exposición de detalles de la infraestructura (rutas de archivos, nombres internos de módulos o bibliotecas, rutas absolutas, etc.).
- Facilitación de ataques de ingeniería social o ataques dirigidos por parte de actores maliciosos.
- Incremento en la superficie de ataque al revelar tecnologías y versiones utilizadas.
- Dificultad para cumplir con normativas de privacidad y protección de datos (como GDPR o HIPAA).

- Incremento de riesgo de explotación de otras vulnerabilidades al proporcionar pistas sobre la lógica interna de la aplicación.
- Pérdida de confianza de usuarios por exposición accidental de información crítica o confidencial ante fallos y errores de ejecución.
- Potencial bloqueo o denegación de servicio si la información filtrada es utilizada para lanzar ataques automatizados.
- Infección de registros de error con datos confidenciales, afectando sistemas de monitoreo o alertamiento.

Buenas prácticas

- Utilizar RAII y tipos que implementen `Drop` para liberar recursos automáticamente incluso ante `panic!`.

```
//-- manejo manual del recurso que puede omitirse en caso de panic
let file = std::fs::File::create("data.txt").unwrap();
if error_detectado {
  panic!("error inesperado");
}
std::fs::remove_file("data.txt").unwrap();
```

```
//-- encapsulación en estructura con Drop para liberar el recurso automáticamente
struct TempFile {
  path: String,
}
impl Drop for TempFile {
  fn drop(&mut self) {
    let _ = std::fs::remove_file(&self.path);
  }
}
let temp = TempFile { path: "data.txt".into() };
let _file = std::fs::File::create(&temp.path).unwrap();
if error_detectado {
  panic!("error inesperado");
}
```

- Propagar errores con `Result` y el operador `?` en lugar de usar `panic!` o `unwrap()` que interrumpan el flujo y omitan liberaciones necesarias.

```
//-- uso de unwrap que puede abortar sin ejecutar lógica de limpieza
let mut conn = conectar_bd().unwrap();
let datos = leer_datos().unwrap();
conn.guardar(datos).unwrap();
conn.cerrar();
```

```
//-- propagación de errores asegurando cierre explícito antes de retornar
fn procesar() -> Result<(), Error> {
  let mut conn = conectar_bd()?;
  let datos = leer_datos()?;
  conn.guardar(datos)?;
  conn.cerrar();
  Ok(())
}
```

- Implementar cancelación segura en tareas asíncronas liberando recursos mediante bloques `Drop` implícitos o guardas de alcance.

```
//-- tarea asíncrona sin garantizar liberación ante cancelación
let socket = abrir_socket().await;
procesar(socket).await;
```

```
//-- uso de guarda de alcance para asegurar cierre del recurso
struct SocketGuard {
  socket: Socket,
}
impl Drop for SocketGuard {
  fn drop(&mut self) {
    let _ = self.socket.cerrar();
  }
}
let guard = SocketGuard { socket: abrir_socket().await };
procesar(&guard.socket).await;
```

2.3.3.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 7.1**: Exception Safety.

2.4 Validaciones

2.4.1 [rust.val.1] - No deserializar directamente datos no confiables a tipos con invariantes complejos

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

La deserialización directa de datos externos puede comprometer las **invariantes de tipo** que garantizan el comportamiento correcto del programa. Los datos no confiables pueden contener valores que violen las restricciones internas de estructuras complejas, provocando estados inconsistentes que deriven en **comportamientos indefinidos** o vulnerabilidades de seguridad durante la ejecución.

Codificación insegura

Se muestra cómo deserializar directamente un JSON externo hacia una estructura que tiene reglas internas, como edad mínima y formato de email. La intención es observar qué ocurre cuando se omiten validaciones y se permite que datos no confiables construyan directamente un objeto que debería mantener invariantes.

```
use serde::Deserialize;
use serde_json;

//-- Se define una estructura con invariantes lógicos: edad mínima y email no vacío
#[derive(Debug, Deserialize)]
struct User {
  email: String,
  age: u8,
}
fn main() {
  //-- Simula un JSON recibido desde una fuente no confiable
  let data = r#"{"email": "", "age": 12}"#;

  //-- Se deserializa directamente al tipo con invariantes complejos
  let user: User = serde_json::from_str(data).unwrap();

  //-- El objeto se crea aunque viola las reglas de negocio
  println!("Usuario deserializado: {:?}", user);
}
```

Codificación recomendada

Se presenta una estrategia donde los datos externos se deserializan primero a una estructura simple sin reglas, y luego se transforman a un tipo de dominio que valida sus

invariantes. La intención es separar la representación externa de la lógica interna y asegurar que sólo se construyan objetos válidos.

```
use serde::Deserialize;
use serde_json;
use std::convert::TryFrom;

//-- Estructura que representa únicamente el formato externo del JSON
#[derive(Debug, Deserialize)]
struct UserDto {
  email: String,
  age: u8,
}
//-- Estructura de dominio con invariantes que deben cumplirse
#[derive(Debug)]
struct User {
  email: String,
  age: u8,
}
//-- Se implementa una conversión validada desde el DTO al tipo de dominio
impl TryFrom<UserDto> for User {
  type Error = String;

  fn try_from(dto: UserDto) -> Result<Self, Self::Error> {
    //-- Validación de invariantes antes de construir el objeto
    if dto.email.trim().is_empty() {
      return Err("El email no puede estar vacío".to_string());
    }
    if dto.age < 18 {
      return Err("La edad debe ser mayor o igual a 18".to_string());
    }
    Ok(User {
      email: dto.email,
      age: dto.age,
    })
  }
}
fn main() {
  //-- Simula un JSON recibido desde una fuente no confiable
  let data = r#"{"email": "", "age": 12}"#;

  //-- Se deserializa primero a un tipo simple sin invariantes
  let dto: UserDto = serde_json::from_str(data).unwrap();

  //-- Se valida antes de construir el objeto de dominio
  match User::try_from(dto) {
    Ok(user) => println!("Usuario válido: {:?}", user),
    Err(e) => println!("Error de validación: {}", e),
  }
```

```
}
```

2.4.1.1 Descripción

La deserialización directa de datos no confiables hacia tipos con invariantes complejos introduce vulnerabilidades significativas en las aplicaciones. Los invariantes son condiciones que deben mantenerse verdaderas durante toda la vida útil de una estructura de datos, como rangos válidos de valores, relaciones entre campos o estados internos consistentes.

Cuando se deserializan datos externos sin validación previa, los atacantes pueden manipular la entrada para crear instancias que violen estos invariantes. Esta violación puede provocar **comportamientos indefinidos**, corrupción de memoria, o permitir la ejecución de código malicioso.

Los tipos con invariantes complejos incluyen estructuras que mantienen índices válidos, colecciones con restricciones de tamaño, tipos que representan estados de máquinas finitas, o cualquier estructura donde la validez de los datos depende de múltiples condiciones interrelacionadas.

El problema se agrava cuando los datos provienen de fuentes externas como archivos, bases de datos, comunicaciones de red, o interfaces de usuario. Estas fuentes pueden estar comprometidas o contener datos malformados intencionalmente.

La mitigación efectiva requiere implementar **capas de validación** antes de la deserialización hacia tipos complejos. Esto incluye el uso de tipos intermedios que no mantengan invariantes, validación explícita de los datos deserializados, y la construcción controlada de instancias finales solo después de verificar todas las condiciones necesarias.

Las bibliotecas de deserialización modernas proporcionan mecanismos para implementar validación personalizada durante el proceso de deserialización, permitiendo rechazar datos inválidos antes de que puedan afectar la integridad del sistema.

Riesgos asociados

- Ejecución de código inseguro o no esperado por la violación de invariantes internas del tipo.
- Vulnerabilidad a ataques de deserialización maliciosa (por ejemplo, exploits de lógica de negocio o bypass de validaciones).
- Corrupción de memoria o comportamiento indefinido si se sobrepasan las garantías de seguridad de Rust.
- Elevación de privilegios al alterar estados internos supuestamente inalcanzables.

- Denegación de servicio por pánico o error lógico al operar con datos inconsistentes.
- Filtración accidental de información sensible mediante estructuras malformadas.
- Introducción de errores lógicos difíciles de detectar en tiempo de compilación.
- Violación de invariantes de sincronización o concurrencia (por ejemplo, data races o bloqueos inesperados).

Buenas prácticas

- Deserializar primero en una estructura intermedia simple y validar explícitamente antes de construir tipos con invariantes.

```
//-- Deserialización directa a un tipo con invariantes internos
#[derive(Deserialize)]
struct Usuario {
  edad: u8,
  email: String,
}
impl Usuario {
  fn new(edad: u8, email: String) -> Result {
    if edad < 18 { return Err("Edad inválida".into()); }
    if !email.contains("@") { return Err("Email inválido".into()); }
    Ok(Self { edad, email })
  }
}
```

```
//-- Deserialización a DTO simple y validación previa a la construcción
#[derive(Deserialize)]
struct UsuarioDto {
  edad: u8,
  email: String,
}
let dto: UsuarioDto = serde_json::from_str(input)?;
let usuario = Usuario::new(dto.edad, dto.email)?;
```

- Evitar exponer campos públicos en tipos con invariantes; usar constructores validados tras la deserialización.

```
//-- Tipo con campos públicos que permite estados inválidos tras deserializar
#[derive(Deserialize)]
pub struct Cuenta {
  pub saldo: i64,
}
```

```
//-- Tipo con campos privados y creación controlada tras validar datos externos
```

```
pub struct Cuenta {
  saldo: i64,
}
impl Cuenta {
  pub fn new(saldo: i64) -> Result {
    if saldo < 0 { return Err("Saldo negativo".into()); }
    Ok(Self { saldo })
  }
}
#[derive(Deserialize)]
struct CuentaDto {
  saldo: i64,
}
```

- Implementar `TryFrom` para convertir desde tipos deserializados a tipos de dominio con validación centralizada.

```
//-- Conversión implícita sin validación desde datos externos
#[derive(Deserialize)]
struct Pedido {
  cantidad: u32,
}
```

```
//-- Conversión explícita usando TryFrom con validaciones de invariantes
struct PedidoSeguro {
  cantidad: u32,
}
impl TryFrom for PedidoSeguro {
  type Error = String;
  fn try_from(p: Pedido) -> Result {
    if p.cantidad == 0 { return Err("Cantidad inválida".into()); }
    Ok(Self { cantidad: p.cantidad })
  }
}
let pedido: Pedido = serde_json::from_str(input)?;
let seguro = PedidoSeguro::try_from(pedido)?;
```

- Aplicar validaciones adicionales posteriores a la deserialización aunque el tipo derive `Deserialize`.

```
//-- Uso directo del objeto deserializado sin verificación adicional
#[derive(Deserialize)]
struct Config {
  timeout: u64,
}
let cfg: Config = serde_json::from_str(input)?;
```

```
usar_timeout(cfg.timeout);
```

```
//-- Verificación explícita de límites antes de utilizar los valores
let cfg: Config = serde_json::from_str(input)?;
if cfg.timeout == 0 || cfg.timeout > 300 {
  return Err("Timeout fuera de rango".into());
}
usar_timeout(cfg.timeout);
```

2.4.1.2 Estándares metodológicos

- **Rust-7.BUILDER**: Builders enable construction of complex values.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 4.4**: Transmutes.

2.4.2 [rust.val.2] - Validar tamaños y longitudes antes de realizar asignaciones dependientes de entrada externa

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

Las asignaciones de memoria y operaciones sobre estructuras de datos que dependen de valores proporcionados por fuentes externas pueden generar **vulnerabilidades críticas** si no se validan adecuadamente los tamaños y longitudes. Los atacantes pueden explotar esta falta de validación para provocar **desbordamientos de buffer**, consumo excesivo de memoria o corrupción de datos, comprometiendo la integridad y disponibilidad del sistema.

Codificación insegura

Programa que lee desde la entrada estándar un tamaño indicado por el usuario y reserva un buffer con esa capacidad. Se busca mostrar cómo una asignación basada directamente en datos externos puede provocar consumo excesivo de memoria o fallos en tiempo de ejecución cuando no se validan límites.

```
use std::io::{self, Read};

//-- Lee un número desde la entrada estándar que representa el tamaño del buffer
fn main() {
  //-- Variable para almacenar la entrada completa
  let mut input = String::new();

  //-- Lee datos desde stdin sin validar contenido ni tamaño
```

```
  io::stdin().read_to_string(&mut input).expect("Error al leer");

  //-- Convierte la entrada en un número sin validar rango razonable
  let size: usize = input.trim().parse().expect("Número inválido");

  //-- Reserva un vector con la capacidad indicada por el usuario
  let mut buffer: Vec = Vec::with_capacity(size);
  //-- Simula uso del buffer
  buffer.resize(size, 0);
  //-- Muestra el tamaño reservado
  println!("Buffer creado con tamaño: {}", buffer.len());
}
```

Codificación recomendada

Programa que lee desde la entrada estándar un tamaño indicado por el usuario y reserva un buffer con esa capacidad, incorporando validaciones de formato y límites máximos permitidos. Se busca aprender a aplicar controles de rango antes de realizar asignaciones dependientes de datos externos para evitar consumo excesivo de recursos.

```
use std::io::{self, Read};

//-- Define un límite máximo permitido para la asignación
const MAX_BUFFER_SIZE: usize = 1024 * 1024; // 1 MB

fn main() {
  //-- Variable para almacenar la entrada completa
  let mut input = String::new();

  //-- Lee datos desde stdin
  io::stdin().read_to_string(&mut input).expect("Error al leer");

  //-- Intenta convertir la entrada en número de forma controlada
  let size: usize = match input.trim().parse() {
    Ok(value) => value,
    Err(_) => {
      println!("Entrada inválida");
      return;
    }
  };

  //-- Valida que el tamaño no exceda el límite permitido
  if size == 0 || size > MAX_BUFFER_SIZE {
    println!("Tamaño fuera de rango permitido");
    return;
  }
  //-- Reserva el buffer solo después de validar
  let mut buffer: Vec = Vec::with_capacity(size);
```

```
    buffer.resize(size, 0);

    //-- Muestra el tamaño reservado de forma segura
    println!("Buffer creado con tamaño: {}", buffer.len());
}
```

2.4.2.1 Descripción

La validación de tamaños y longitudes constituye un control necesario antes de realizar asignaciones que dependen de datos externos no confiables. Esta práctica previene la asignación de recursos basada en valores potencialmente maliciosos o incorrectos que podrían comprometer la estabilidad y seguridad del programa.

Los datos externos incluyen entrada de usuario, contenido de archivos, respuestas de red, argumentos de línea de comandos y cualquier información que provenga de fuentes no controladas por el programa. Estos datos pueden contener valores extremos, negativos cuando se esperan positivos, o magnitudes que excedan los límites razonables del sistema.

La ausencia de validación previa permite que atacantes o datos corruptos provoquen **agotamiento de memoria**, **degradación del rendimiento** o **fallos del sistema**. Los valores no validados pueden resultar en asignaciones excesivas que consuman recursos disponibles o generen comportamientos impredecibles en estructuras de datos.

El sistema de tipos y las verificaciones en tiempo de compilación proporcionan protecciones básicas, pero no pueden anticipar los valores específicos que llegarán durante la ejecución. La validación explícita de rangos y límites máximos actúa como una barrera adicional contra datos malformados o maliciosos.

La mitigación efectiva requiere establecer **límites máximos razonables** para tamaños y longitudes basados en los requisitos reales del programa y los recursos disponibles del sistema. La validación debe ocurrir inmediatamente después de recibir los datos externos y antes de utilizarlos en operaciones de asignación o dimensionamiento.

Las técnicas de mitigación incluyen la verificación de rangos numéricos, la imposición de límites superiores para colecciones y buffers, y la validación de coherencia entre múltiples valores relacionados. El manejo de errores debe ser explícito cuando los valores exceden los límites establecidos.

Riesgos asociados

- Desbordamiento de buffer (buffer overflow) por intentar copiar o escribir más datos de los que la estructura o variable puede alojar.
- Condición de pánico o crash de la aplicación debido a la violación de invariantes de tamaño en arrays, vectores o cadenas.
- Acceso fuera de los límites (out-of-bounds) que puede provocar errores o

comportamientos indefinidos, aunque Rust usa comprobaciones en tiempo de ejecución, pueden ocurrir errores lógicos o panics.

- Inyección de datos maliciosos que provoquen sobreescritura o corrupción de memoria y comprometan la lógica de la aplicación.
- Denegación de servicio (DoS) al agotar recursos por tratar de asignar grandes bloques de memoria sin validación previa.
- Filtración de información sensible por lectura o escritura incorrecta en estructuras de datos asociadas a la memoria recibida.
- Vulnerabilidades en bibliotecas externas llamadas desde código Rust (por ejemplo, FFI con C), donde no existen las mismas garantías de seguridad y la falta de validación puede resultar fatal.
- Exposición a ataques tipo integer overflow/underflow en el cálculo de tamaños si los parámetros no son validados adecuadamente antes de asignar memoria.

Buenas prácticas

- Validar explícitamente la longitud de datos externos antes de realizar asignaciones dinámicas.

```
//-- Asignación directa basada en longitud proporcionada externamente sin
validación
  fn procesar(longitud: usize) {
    let buffer = vec![0u8; longitud];
    println!("Buffer creado: {}", buffer.len());
  }
```

```
//-- Validación de límites máximos antes de realizar la asignación
  fn procesar(longitud: usize) {
    const MAX_TAM: usize = 1024;
    if longitud > MAX_TAM {
      return;
    }
    let buffer = vec![0u8; longitud];
    println!("Buffer creado: {}", buffer.len());
  }
```

- Comprobar la longitud mínima y máxima antes de convertir o copiar datos en estructuras tipadas.

```
//-- Conversión directa de entrada a array fijo sin verificar longitud
  fn convertir(datos: &[u8]) {
    let array: [u8; 8] = datos.try_into().unwrap();
    println!("{:?}", array);
```

```
}
```

```
//-- Validación explícita de longitud antes de la conversión
  fn convertir(datos: &[u8]) {
    if datos.len() != 8 {
      return;
    }
    let array: [u8; 8] = datos.try_into().unwrap();
    println!("{:?}", array);
  }
```

- Validar tamaños antes de reservar capacidad con `with_capacity` para evitar consumo excesivo de memoria.

```
//-- Reserva de memoria basada en valor externo sin control de límites
  fn reservar(capacidad: usize) {
    let mut datos = Vec::with_capacity(capacidad);
    datos.push(1);
  }
```

```
//-- Validación previa del tamaño máximo permitido antes de reservar
  fn reservar(capacidad: usize) {
    const LIMITE: usize = 2048;
    if capacidad > LIMITE {
      return;
    }
    let mut datos = Vec::with_capacity(capacidad);
    datos.push(1);
  }
```

2.4.2.2 Estándares metodológicos

- **Rust-6.GENERIC**: Functions minimize assumptions about parameters by using generics.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 9.2**: Allocating.

2.4.3 [rust.val.3] - No confiar en derives automáticos cuando puedan violar invariantes de seguridad

ID	**rust.val.3**	**Uso**	**Recomendado**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los `derive` automáticos generan implementaciones de traits sin considerar las restricciones específicas del tipo de dato. Esta generación automática puede crear comportamientos que **comprometen invariantes de seguridad** establecidas en el diseño del código, especialmente en tipos que manejan datos sensibles o mantienen estados críticos del sistema.

Codificación insegura

Se define un tipo que representa una clave secreta con la intención de mantener su confidencialidad y evitar copias innecesarias en memoria. El objetivo es mostrar cómo ciertas implementaciones automáticas pueden romper invariantes de seguridad como la no exposición del secreto o la restricción de copias.

```
use std::fmt;

//-- Estructura que representa una clave secreta
#[derive(Clone, Debug)]
struct SecretKey {
  //-- Contenido sensible que no debería copiarse ni mostrarse
  key: String,
}
fn main() {
  //-- Se crea una clave secreta
  let original = SecretKey {
    key: String::from("SUPER_SECRETO"),
  };

  //-- Se realiza una copia completa del secreto en memoria
  let copia = original.clone();

  //-- Se imprime el contenido sensible en logs
  println!("Clave original: {:?}", original);
  println!("Clave copia: {:?}", copia);
}
```

Codificación recomendada

Se define un tipo que representa una clave secreta manteniendo como invariante que su contenido no pueda clonarse ni mostrarse accidentalmente. Se implementan

manualmente los comportamientos necesarios para evitar la exposición del dato sensible y reducir riesgos de fuga de información.

```
use std::fmt;

//-- Estructura que representa una clave secreta
struct SecretKey {
  //-- Contenido sensible que no debe exponerse
  key: String,
}
//-- Implementación manual de Debug que oculta el secreto
impl fmt::Debug for SecretKey {
  fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
    //-- Se evita mostrar el valor real del secreto
    write!(f, "SecretKey {{ key: ***REDACTED*** }}")
  }
}
impl SecretKey {
  //-- Constructor controlado
  fn new(value: &str) -> Self {
    SecretKey {
      key: value.to_string(),
    }
  }
  //-- Método que permite usar el secreto sin exponerlo
  fn authenticate(&self) -> bool {
    //-- Simulación de validación interna
    !self.key.is_empty()
  }
}
fn main() {
  //-- Se crea una clave secreta
  let secret = SecretKey::new("SUPER_SECRETO");

  //-- No es posible clonar la clave porque no implementa Clone
  //-- let copia = secret.clone(); // Esto no compilaría

  //-- La impresión no revela el contenido sensible
  println!("Clave: {:?}", secret);

  //-- Uso controlado del secreto
  println!("Autenticación válida: {}", secret.authenticate());
}
```

2.4.3.1 Descripción

El sistema de `derive` opera a nivel sintáctico, aplicando patrones predefinidos a todos los campos de una estructura o enumeración. Esta aproximación mecánica no tiene

conocimiento de las restricciones semánticas que pueden existir en el diseño del tipo, como rangos válidos, estados internos consistentes, o niveles de acceso diferenciados.

Los riesgos principales incluyen la **exposición inadvertida de información confidencial** a través de traits como `Debug` o `Display`, la **violación de invariantes de estado** mediante clonación o comparación automática, y la **creación de vulnerabilidades de serialización** cuando se derivan traits relacionados con la persistencia de datos.

La mitigación efectiva requiere una **evaluación explícita de cada trait derivado** en el contexto de los requisitos de seguridad del tipo. Esto incluye la implementación manual de traits cuando sea necesario aplicar lógica específica de validación, filtrado o transformación. También implica el uso de atributos de control granular para excluir campos sensibles de las implementaciones automáticas cuando sea posible.

La práctica segura consiste en **derivar únicamente los traits que no comprometan las invariantes del sistema** y proporcionar implementaciones personalizadas para aquellos que requieran consideraciones especiales de seguridad. Esta aproximación mantiene el control sobre el comportamiento del código mientras preserva la integridad de los datos y la confidencialidad de la información sensible.

Riesgos asociados

- Ejecución de código inseguro debido a la implementación automática de traits que no respetan restricciones de seguridad.
- Divulgación de información sensible a través de traits como `Debug` o `Clone` en tipos que contienen datos secretos.
- Ruptura de invariantes críticas de seguridad por el comportamiento por omisión de derives como `PartialEq`, `Eq`, `Hash` o `Ord`, abriendo la puerta a errores lógicos y vulnerabilidades.
- Generación automática de código que permite duplicar, copiar o comparar estructuras que requieren control explícito y validaciones adicionales.
- Incremento de la superficie de ataque por exposición involuntaria de funcionalidades susceptibles de abuso por parte de un atacante.
- Dificultad para detectar y auditar fallos de seguridad, ya que los derives automáticos pueden ser menos evidentes durante la revisión del código.

Buenas prácticas

- Evitar `#[derive(Clone)]` en estructuras que gestionan secretos o recursos sensibles si la copia puede exponer información.

```
//-- Derivación automática que permite duplicar datos sensibles
#[derive(Clone)]
struct Credencial {
```

```
  token: String,
}
```

```
//-- Implementación controlada que evita la clonación del secreto
struct Credencial {
  token: String,
}
impl Credencial {
  fn new(token: String) -> Self {
    Self { token }
  }
}
```

- No utilizar `#[derive(Debug)]` en estructuras que contengan información confidencial para evitar filtraciones en logs.

```
//-- Derivación automática que expone campos sensibles en logs
#[derive(Debug)]
struct Usuario {
  nombre: String,
  password: String,
}
```

```
//-- Implementación personalizada que omite datos sensibles en el formateo
struct Usuario {
  nombre: String,
  password: String,
}
impl core::fmt::Debug for Usuario {
'_>) -> core::fmt::Result {
    f.debug_struct("Usuario")
      .field("nombre", &self.nombre)
      .field("password", &"***")
      .finish()
  }
}
```

- Evitar `#[derive(PartialEq, Eq)]` cuando se requiera comparación en tiempo constante para prevenir ataques de temporización.

```
//-- Derivación automática que usa comparación estándar no constante
#[derive(PartialEq, Eq)]
struct ApiKey([u8; 32]);
```

```
//-- Implementación manual con comparación en tiempo constante
struct ApiKey([u8; 32]);
```

```
impl ApiKey {
  fn equals(&self, other: &Self) -> bool {
    use subtle::ConstantTimeEq;
    self.0.ct_eq(&other.0).into()
  }
}
```

2.4.3.2 Estándares metodológicos

- **Rust-2.COMMON-TRAITS**: Types eagerly implement common traits.
- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rustonomicon 1.1**: How Safe and Unsafe Interact.

2.5 Gestión de errores

2.5.1 [rust.err.1] - No usar unwrap, expect o panic en código productivo donde el fallo sea posible

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

Los métodos `unwrap()`, `expect()` y la macro `panic!()` terminan abruptamente la ejecución del programa cuando encuentran errores o condiciones inesperadas. Su uso en código productivo introduce **puntos de fallo no controlados** que pueden provocar caídas del sistema en situaciones donde el error podría ser manejado de forma segura. Esta práctica compromete la estabilidad y disponibilidad de las aplicaciones.

Codificación insegura

Lectura de un archivo de texto cuyo contenido se muestra por consola. La intención es ilustrar cómo manejar el resultado de operaciones que pueden fallar, como abrir o leer un archivo, y qué ocurre cuando no se gestiona correctamente ese posible error.

```
use std::fs::File;
use std::io::Read;

fn main() {
  //-- Se intenta abrir un archivo que puede no existir
  let mut file = File::open("config.txt").unwrap();

  //-- Se crea un buffer para almacenar el contenido
  let mut contents = String::new();
```

```
  //-- Se intenta leer el contenido completo del archivo
  file.read_to_string(&mut contents).unwrap();

  //-- Se muestra el contenido leído
  println!("Contenido: {}", contents);
}
```

Codificación recomendada

Lectura de un archivo de texto cuyo contenido se muestra por consola, gestionando de forma explícita los posibles errores. Se pretende aprender a propagar y tratar fallos sin abortar la ejecución de forma inesperada, permitiendo un control adecuado del flujo del programa.

```
use std::fs::File;
use std::io::{self, Read};
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
  //-- Se intenta abrir un archivo y se propaga el error si ocurre
  let mut file = match File::open("config.txt") {
    Ok(f) => f,
    Err(e) => {
      //-- Se informa del error y se retorna de forma controlada
      eprintln!("No se pudo abrir el archivo: {}", e);
      return Err(Box::new(e));
    }
  };

  //-- Buffer para almacenar el contenido del archivo
  let mut contents = String::new();

  //-- Se intenta leer el archivo y se propaga el posible error
  file.read_to_string(&mut contents)?;

  //-- Se muestra el contenido leído si todo fue correcto
  println!("Contenido: {}", contents);

  Ok(())
}
```

2.5.1.1 Descripción

Los métodos `unwrap()`, `expect()` y la macro `panic!()` provocan la terminación inmediata del programa cuando encuentran condiciones de error. Esta terminación abrupta puede generar vulnerabilidades de seguridad, especialmente en aplicaciones que manejan datos sensibles o requieren alta disponibilidad.

El uso de estos métodos en código productivo introduce **puntos de fallo no controlados** que pueden ser explotados por atacantes para provocar denegaciones de servicio. Cuando el programa termina inesperadamente, puede dejar recursos sin liberar, transacciones incompletas o datos en estados inconsistentes.

La terminación forzosa del programa también puede exponer información sensible a través de mensajes de pánico o volcados de memoria. Los atacantes pueden aprovechar estos fallos para obtener detalles sobre la estructura interna del programa o causar interrupciones en servicios críticos.

La mitigación de estos riesgos se basa en el **manejo explícito de errores** mediante el sistema de tipos. Las funciones que pueden fallar deben retornar tipos `Result` u `Option`, permitiendo que el código cliente decida cómo proceder ante cada situación de error. Este enfoque mantiene el control del flujo de ejecución y permite implementar estrategias de recuperación apropiadas.

En casos excepcionales donde la terminación del programa sea la única respuesta válida, se debe documentar claramente esta decisión y considerar mecanismos de logging que no comprometan la seguridad del sistema.

Riesgos asociados

- Interrupción inesperada de la aplicación debido a panics en tiempo de ejecución.
- Pérdida de datos o estado de la aplicación al no manejar errores de forma controlada.
- Imposibilidad de ofrecer mensajes de error claros y manejables al usuario o a otros sistemas.
- Exposición potencial de información sensible en mensajes de `panic` o `expect`.
- Dificultad para garantizar la disponibilidad y robustez del sistema ante situaciones inesperadas.
- Mayores probabilidades de provocar denegaciones de servicio (DoS), especialmente en aplicaciones concurrentes o servicios críticos.
- Imposibilidad de implementar políticas de recuperación ante errores o validaciones adicionales.
- Complicaciones para realizar trazabilidad y monitoreo eficiente de fallos en producción.

Buenas prácticas

- Propagar errores utilizando `Result` y el operador `?` en lugar de abortar la ejecución.

```
//-- uso de unwrap que aborta ante error
```

```
fn read_config() -> String {
  let content = std::fs::read_to_string("config.toml").unwrap();
  content
}
```

```
//-- propagación controlada del error al llamador
fn read_config() -> Result<String, std::io::Error> {
  let content = std::fs::read_to_string("config.toml")?;
  Ok(content)
}
```

- Gestionar explícitamente los casos `None` o `Err` mediante `match` o `if let` para aplicar lógica segura.

```
//-- uso de expect que finaliza el proceso si no existe valor
fn get_user_name(id: Option<u32>) -> u32 {
  id.expect("id requerido")
}
```

```
//-- manejo explícito del caso None con respuesta controlada
fn get_user_name(id: Option<u32>) -> Result<u32, String> {
  match id {
    Some(value) => Ok(value),
    None => Err("id no proporcionado".to_string()),
  }
}
```

2.5.1.2 Estándares metodológicos

- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rust-8.DTOR-FAIL**: Destructors never fail.

2.5.2 [rust.err.2] - Representar fallos recuperables exclusivamente mediante Result u Option

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

Los fallos recuperables requieren un manejo explícito que permita al código continuar su ejecución tras evaluar el error. El uso de mecanismos inadecuados para representar estos fallos puede generar **comportamientos indefinidos**, **terminaciones inesperadas** o **estados inconsistentes** en la aplicación. La representación incorrecta de errores

recuperables compromete la robustez del programa y dificulta el mantenimiento del código.

Codificación insegura

Se muestra una forma de manejar un posible error al convertir una cadena a número entero. La intención es evidenciar cómo un fallo recuperable puede tratarse de forma abrupta en tiempo de ejecución, impidiendo al programa reaccionar de manera controlada ante entradas inválidas.

```
use std::env;

fn parse_number(input: &str) -> i32 {
  //-- intento de conversión que puede fallar
  input.parse::<i32>().unwrap()
}
fn main() {
  //-- se obtiene el primer argumento de la línea de comandos
  let arg = env::args().nth(1).unwrap();

  //-- si la conversión falla, el programa termina abruptamente
  let number = parse_number(&arg);

  //-- uso normal del valor convertido
  println!("Número duplicado: {}", number * 2);
}
```

Codificación recomendada

Se aborda el mismo escenario de conversión de cadena a número entero, pero representando el posible fallo exclusivamente mediante `Result`. El objetivo es propagar y gestionar el error de forma explícita, permitiendo al programa responder de manera controlada ante entradas inválidas.

```
use std::env;
use std::num::ParseIntError;

fn parse_number(input: &str) -> Result<i32, ParseIntError> {
  //-- la función devuelve un Result en lugar de abortar
  input.parse::<i32>()
}
fn main() -> Result<(), Box<dyn std::error::Error>> {
  //-- se obtiene el primer argumento de forma controlada
  let arg = env::args()
    .nth(1)
    .ok_or("Debe proporcionar un número como argumento")?;

  //-- se propaga el posible error de conversión usando ?
```

```
    let number = parse_number(&arg)?;

    //-- uso normal del valor convertido
    println!("Número duplicado: {}", number * 2);

    //-- finalización correcta
    Ok(())
}
```

2.5.2.1 Descripción

Los fallos recuperables son aquellos errores de los que un programa puede recuperarse y continuar su ejecución de manera controlada. Estos incluyen operaciones que pueden fallar por condiciones externas como archivos inexistentes, conexiones de red interrumpidas, datos de entrada inválidos o recursos temporalmente no disponibles.

La representación de estos fallos debe realizarse exclusivamente mediante los tipos `Result<T, E>` y `Option<T>`. El tipo `Result` se utiliza cuando la operación puede fallar con información específica sobre el error, mientras que `Option` se emplea cuando la ausencia de un valor constituye el único tipo de fallo posible.

Esta aproximación contrasta con el uso de **panic**, que está reservado para errores irrecuperables donde el programa no puede continuar de forma segura. Los panics interrumpen abruptamente la ejecución y desenrollan la pila, comportamiento inadecuado para errores de los que el programa debería recuperarse.

El manejo inadecuado de fallos recuperables puede provocar **terminaciones inesperadas del programa**, pérdida de datos, estados inconsistentes y experiencias de usuario deficientes. Los errores no controlados pueden propagarse de manera impredecible, dificultando el diagnóstico y la resolución de problemas.

La mitigación se basa en el **diseño explícito de interfaces** que declaren claramente qué operaciones pueden fallar y de qué manera. Esto incluye la definición de tipos de error específicos, el uso consistente de `Result` y `Option` en las firmas de funciones, y la implementación de estrategias de manejo de errores apropiadas para cada contexto de la aplicación.

El sistema de tipos garantiza que los errores no puedan ser ignorados accidentalmente, forzando a los desarrolladores a tomar decisiones explícitas sobre cómo manejar cada posible fallo. Esta aproximación mejora la **robustez y predictibilidad** del software al hacer visible el flujo de control de errores en tiempo de compilación.

Riesgos asociados

- Pérdida de control en el manejo de errores, llevando a caídas inesperadas (panic) de la aplicación.
- Fugas de información potencialmente sensible por el uso indebido de

mecanismos inseguros para representar errores.

- Dificultad para diferenciar errores recuperables de errores fatales, derivando en respuestas inapropiadas ante fallos.
- Incremento en la probabilidad de condiciones de carrera o inconsistencias de estado al no gestionar adecuadamente los fallos durante la ejecución.
- Reducción en la mantenibilidad y legibilidad del código, facilitando la introducción accidental de nuevos errores de lógica.
- Imposibilidad de realizar un manejo centralizado y predecible de los errores, lo que puede afectar la robustez y confiabilidad del sistema.
- Complicaciones en la interoperabilidad y testeo, ya que las APIs no comunican claramente cuándo y cómo pueden fallar.

Buenas prácticas

- Evitar el uso de `panic!` para errores recuperables y devolver `Result` en su lugar.

```
//-- uso de panic ante error recuperable
fn dividir(a: i32, b: i32) -> i32 {
  if b == 0 {
    panic!("división por cero");
  }
  a / b
}
```

```
//-- devolución de Result para manejo explícito del error
fn dividir(a: i32, b: i32) -> Result<i32, String> {
  if b == 0 {
    return Err("división por cero".to_string());
  }
  Ok(a / b)
}
```

- No utilizar valores centinela para indicar ausencia de datos; emplear `Option`.

```
//-- uso de valor centinela para indicar ausencia
fn buscar_usuario(id: u32) -> String {
  if id == 0 {
    return "".to_string();
  }
  "usuario".to_string()
}
```

```
//-- uso de Option para representar ausencia explícita
fn buscar_usuario(id: u32) -> Option<String> {
```

```
  if id == 0 {
    return None;
  }
  Some("usuario".to_string())
}
```

- Propagar errores con el operador `?` en lugar de ignorarlos o forzar `unwrap()`.

```
//-- uso de unwrap que puede abortar la ejecución
fn leer_config() -> String {
  let contenido = std::fs::read_to_string("config.txt").unwrap();
  contenido
}
```

```
//-- propagación del error mediante Result y ?
fn leer_config() -> Result<String, std::io::Error> {
  let contenido = std::fs::read_to_string("config.txt")?;
  Ok(contenido)
}
```

2.5.2.2 Estándares metodológicos

- **Rust-2.GOOD-ERR**: Error types are meaningful and well-behaved.
- **Rust-7.CUSTOM-TYPE**: Arguments convey meaning through types, not bool or Option.

2.5.3 [rust.err.3] - No ignorar valores Result marcados como must_use

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

Los valores de tipo `Result` marcados con el atributo `must_use` requieren manejo explícito por parte del programador. **Ignorar estos valores puede ocultar errores críticos** y generar comportamientos impredecibles en tiempo de ejecución, comprometiendo la robustez del programa.

Codificación insegura

Se muestra un programa que crea un archivo y escribe contenido en él. La intención es observar qué ocurre cuando las funciones que devuelven `Result` no son gestionadas, ignorando posibles errores de E/S. El objetivo es entender el riesgo de perder información relevante sobre fallos en tiempo de ejecución.

```
use std::fs::File;
use std::io::Write;

fn main() {
  //-- Se intenta crear un archivo sin gestionar el posible error
  File::create("salida.txt");

  //-- Se intenta abrir el archivo nuevamente
  let mut file = File::create("salida.txt").unwrap();

  //-- Se escribe en el archivo, pero se ignora el Result devuelto
  file.write_all(b"Contenido de prueba");

  //-- El programa finaliza sin comprobar si las operaciones tuvieron éxito
  println!("Proceso finalizado");
}
```

Codificación recomendada

Se muestra un programa que crea un archivo y escribe contenido en él gestionando correctamente los valores `Result`. La intención es propagar y tratar los posibles errores de forma explícita, asegurando que cualquier fallo sea detectado y manejado adecuadamente.

```
use std::fs::File;
use std::io::Write;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
  //-- Se crea el archivo y se propaga el error si ocurre
  let mut file = File::create("salida.txt")?;

  //-- Se escribe en el archivo y se valida el resultado
  file.write_all(b"Contenido de prueba")?;

  //-- Solo se muestra el mensaje si todas las operaciones fueron exitosas
  println!("Proceso finalizado correctamente");

  //-- Se retorna Ok para indicar ejecución satisfactoria
  Ok(())
}
```

2.5.3.1 Descripción

Los tipos `Result` marcados con el atributo `must_use` requieren manejo explícito de su valor de retorno. Ignorar estos valores elimina la capacidad de detectar y responder adecuadamente a condiciones de error, comprometiendo la robustez del programa.

El sistema de tipos utiliza `must_use` para señalar operaciones donde el valor de retorno contiene información crítica sobre el éxito o fallo de la operación. Las funciones que pueden fallar y cuyo fallo debe ser manejado obligatoriamente implementan esta anotación para prevenir el manejo inadecuado de errores.

Cuando se ignora un `Result` marcado como `must_use`, el compilador genera advertencias, pero el código continúa ejecutándose sin verificar si la operación fue exitosa. Esta situación puede provocar **estados inconsistentes** en el programa, donde se asume que una operación fue exitosa cuando en realidad falló.

Los riesgos incluyen corrupción de datos, comportamiento impredecible del programa, y vulnerabilidades de seguridad derivadas de operaciones fallidas no detectadas. Las operaciones de entrada/salida, asignación de memoria, y validación de datos son particularmente susceptibles a estos problemas.

La mitigación requiere manejo explícito de todos los valores `Result` marcados como `must_use`. Esto incluye el uso de métodos como `unwrap()`, `expect()`, coincidencia de patrones, o propagación del error mediante el operador `?`. En casos donde el error puede ignorarse de forma segura, debe utilizarse un método explícito que documente esta decisión intencionada.

El manejo adecuado permite implementar estrategias de recuperación, registro de errores, y terminación controlada del programa cuando sea necesario, manteniendo la integridad del sistema ante condiciones de fallo.

Riesgos asociados

- Pérdida silenciosa de errores importantes durante la ejecución, dificultando la detección y el manejo de fallos.
- Fallos en la lógica del programa debido a la falta de verificación del resultado de operaciones críticas.
- Vulnerabilidad ante comportamientos inesperados que podrían provocar corrupción de datos o caída de la aplicación.
- Superficie de ataque ampliada, ya que se ignoran posibles señales de error generadas por dependencias externas.
- Dificultad para mantener y depurar el software, pues no quedan evidencias explícitas de errores que debieron tratarse.
- Posibilidad de efectos secundarios indeseados si el valor retornado incluye información de control o recursos que no se gestionan adecuadamente.

Buenas prácticas

- Gestionar explícitamente los valores `Result` marcados como `must_use` utilizando `match` o `if let` para evitar ignorar errores.

```
//-- Se ignora el Result devuelto por la operación
std::fs::File::open("config.txt");
```

```
//-- Se maneja explícitamente el Result y se actúa ante error
if let Err(e) = std::fs::File::open("config.txt") {
  eprintln!("Error al abrir archivo: {}", e);
}
```

- Propagar errores usando el operador `?` cuando la función también devuelve `Result`, evitando descartar el valor.

```
//-- Se ignora el resultado dentro de una función que retorna Result
fn cargar() -> Result<(), std::io::Error> {
  std::fs::read_to_string("config.txt");
  Ok(())
}
```

```
//-- Se propaga el error correctamente con el operador ?
fn cargar() -> Result<(), std::io::Error> {
  std::fs::read_to_string("config.txt")?;
  Ok(())
}
```

- Evitar el uso indiscriminado de `unwrap()` o `expect()` en producción y manejar adecuadamente el `Result`.

```
//-- Se fuerza el valor y puede provocar pánico en ejecución
let contenido = std::fs::read_to_string("config.txt").unwrap();
```

```
//-- Se maneja el error de forma controlada
let contenido = match std::fs::read_to_string("config.txt") {
  Ok(c) => c,
  Err(e) => {
    eprintln!("Error al leer archivo: {}", e);
    return;
  }
};
```

- Asignar el `Result` a una variable o usarlo explícitamente cuando la intención sea evaluar el resultado, evitando advertencias por `must_use`.

```
//-- Se descarta el Result sin evaluación
std::env::set_current_dir("/tmp");
```

```
//-- Se evalúa el resultado antes de continuar
```

```
let resultado = std::env::set_current_dir("/tmp");
if resultado.is_err() {
  eprintln!("No se pudo cambiar el directorio");
}
```

2.5.3.2 Estándares metodológicos

- **Rust-2.GOOD-ERR**: Error types are meaningful and well-behaved.
- **Rust-8.VALIDATE**: Functions validate their arguments.

2.5.4 [rust.err.4] - No permitir que un panic cruce límites FFI

ID	rust.err.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los `panic` no deben propagarse a través de las fronteras de `FFI` (Foreign Function Interface) ya que pueden causar **comportamiento indefinido** cuando el código externo no está preparado para manejar el mecanismo de desenrollado de pila. Esta situación genera **inestabilidad del programa** y potenciales fallos de seguridad al interactuar con bibliotecas escritas en otros lenguajes que no comprenden la semántica de los `panic`.

Codificación insegura

Exposición de una función con ABI C que puede generar un `panic` interno no controlado. La intención es mostrar cómo una condición de error provoca un `panic` que atraviesa el límite FFI, lo que genera comportamiento indefinido cuando es invocada desde otro lenguaje o desde un contexto externo.

```
use std::ffi::CStr;
use std::os::raw::c_char;

//-- Función expuesta mediante FFI con ABI C
#[no_mangle]
pub extern "C" fn procesar_cadena(ptr: *const c_char) -> usize {
  //-- Conversión insegura del puntero C a &str
  let c_str = unsafe { CStr::from_ptr(ptr) };

  //-- unwrap() puede provocar panic si no es UTF-8 válido
  let cadena = c_str.to_str().unwrap();

  //-- Se retorna la longitud de la cadena
  cadena.len()
}
fn main() {
  //-- Simulación de llamada externa (como si fuera desde C)
```

```
  let datos = b"\xff\0";
  let ptr = datos.as_ptr() as *const c_char;

  //-- Llamada a la función FFI; si ocurre panic cruza el límite FFI
  let _ = unsafe { procesar_cadena(ptr) };
}
```

Codificación recomendada

Exposición de una función con ABI C que gestiona internamente los errores y captura cualquier `panic` antes de cruzar el límite FFI. Se muestra cómo usar `std::panic::catch_unwind` para garantizar que la función devuelva un código de error en lugar de propagar el `panic` fuera del entorno Rust.

```
use std::ffi::CStr;
use std::os::raw::c_char;
use std::panic;

//-- Función expuesta mediante FFI con control explícito de panic
#[no_mangle]
pub extern "C" fn procesar_cadena(ptr: *const c_char) -> isize {
  //-- Captura de posibles panic para evitar que crucen el límite FFI
  let resultado = panic::catch_unwind(|| {
    //-- Validación básica del puntero
    if ptr.is_null() {
      return Err(());
    }
    //-- Conversión segura del puntero C
    let c_str = unsafe { CStr::from_ptr(ptr) };
    let cadena = c_str.to_str().map_err(|_| ())?;

    Ok(cadena.len() as isize)
  });

  //-- Manejo del resultado sin propagar panic
  match resultado {
    Ok(Ok(longitud)) => longitud,
    _ => -1,
  }
}
fn main() {
  //-- Simulación de llamada externa (como si fuera desde C)
  let datos = b"\xff\0";
  let ptr = datos.as_ptr() as *const c_char;

  //-- La función devuelve -1 en caso de error, sin panic
  let resultado = unsafe { procesar_cadena(ptr) };
  println!("Resultado: {}", resultado);
```

```
}
```

2.5.4.1 Descripción

Los límites de la Interfaz de Función Extranjera (FFI) representan puntos críticos donde el código interactúa con bibliotecas externas o sistemas escritos en otros lenguajes. Cuando un `panic` atraviesa estos límites, se produce un comportamiento indefinido que puede comprometer la estabilidad del programa y la integridad de los datos.

Un `panic` que escapa a través de FFI puede corromper el estado interno de bibliotecas externas que no están diseñadas para manejar este mecanismo de desenrollado de pila. Las bibliotecas escritas en C o C++ asumen un modelo de manejo de errores diferente y no pueden procesar adecuadamente las excepciones que provienen del desenrollado de pila.

El comportamiento indefinido resultante puede manifestarse de múltiples formas: desde terminaciones inesperadas del programa hasta corrupción silenciosa de memoria. Esta situación es particularmente peligrosa porque los efectos pueden no ser inmediatamente visibles, dificultando la detección y depuración de problemas.

La **prevención de este problema** requiere implementar barreras de contención en los puntos de interfaz. Las funciones que actúan como límites FFI deben capturar cualquier `panic` potencial y convertirlo en un mecanismo de manejo de errores compatible con el código externo. Esto incluye el uso de técnicas de captura de pánico y la conversión a códigos de error o valores de retorno apropiados.

Las funciones `extern` que pueden ser llamadas desde código externo constituyen puntos de entrada críticos donde debe aplicarse esta protección. Igualmente, las llamadas salientes hacia código externo deben estar protegidas para evitar que cualquier `panic` interno escape hacia bibliotecas que no pueden manejarlo adecuadamente.

Riesgos asociados

- Corrupción de memoria en el código no Rust debido a la interrupción abrupta del flujo de ejecución.
- Fugas de memoria por no liberar correctamente los recursos al atravesar el límite FFI.
- Resultados indefinidos o comportamiento inesperado en bibliotecas externas al propagarse el `panic`.
- Degradación de la estabilidad de la aplicación por falta de mecanismos de manejo de errores compatibles con otros lenguajes.
- Pérdida de información relevante para el diagnóstico de fallos, ya que el stack unwinding puede no estar soportado fuera de Rust.

- Favorcimiento de vulnerabilidades de seguridad aprovechables por atacantes externos que provoquen panics intencionados.
- Incompatibilidad con las expectativas de los lenguajes anfitriones (C, C++, etc.), que normalmente no soportan excepciones no gestionadas.
- Dificultad para depurar e identificar el origen de errores interlinguaje.

Buenas prácticas

- Encapsular el código FFI con `std::panic::catch_unwind` para evitar que un `panic!` cruce el límite externo.

```
//-- panic puede propagarse a través del límite FFI
#[no_mangle]
pub extern "C" fn procesar(valor: i32) -> i32 {
  if valor < 0 {
    panic!("valor inválido");
  }
  valor * 2
}
```

```
//-- uso de catch_unwind para capturar panic antes de cruzar FFI
#[no_mangle]
pub extern "C" fn procesar(valor: i32) -> i32 {
  let resultado = std::panic::catch_unwind(|| {
    if valor < 0 {
      panic!("valor inválido");
    }
    valor * 2
  });
  match resultado {
    Ok(v) => v,
    Err(_) => -1,
  }
}
```

- Definir contratos de error explícitos en funciones `extern "C"`, evitando el uso de `unwrap()` o `expect()` dentro del límite FFI.

```
//-- uso de unwrap que puede provocar panic en límite FFI
#[no_mangle]
pub extern "C" fn longitud(ptr: *const u8, len: usize) -> usize {
  let slice = unsafe { std::slice::from_raw_parts(ptr, len) };
  let texto = std::str::from_utf8(slice).unwrap();
  texto.len()
}
```

```
//-- validación explícita y retorno de código de error controlado
#[no_mangle]
pub extern "C" fn longitud(ptr: *const u8, len: usize) -> isize {
  let slice = unsafe { std::slice::from_raw_parts(ptr, len) };
  match std::str::from_utf8(slice) {
    Ok(texto) => texto.len() as isize,
    Err(_) => -1,
  }
}
```

2.5.4.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rustonomicon 12.1**: #[panic_handler].
- **Rustonomicon 7.1**: Exception Safety.

2.5.5 [rust.err.5] - Garantizar que Drop mantiene invariantes incluso ante panic

ID	rust.err.5	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La implementación del trait `Drop` puede ejecutarse en contextos donde se han producido pánicos, creando situaciones donde los invariantes del tipo pueden violarse. Cuando el destructor no maneja adecuadamente estos escenarios, puede generar **estados inconsistentes** o **corrupción de datos** que comprometen la integridad del programa.

Codificación insegura

Gestión de un recurso que debe cerrarse correctamente al finalizar su uso. Se busca mostrar cómo un destructor puede romper invariantes y provocar un doble pánico si asume estados internos válidos sin verificarlos, especialmente cuando se ejecuta durante el desenrollado de la pila por un `panic`.

```
use std::fs::File;
use std::io::Write;

struct TempFile {
  file: Option,
}
impl TempFile {
  fn new(path: &str) -> Self {
    //-- Se crea el archivo y se guarda dentro de Option
    let mut f = File::create(path).unwrap();
```

```
    writeln!(f, "datos iniciales").unwrap();
    TempFile { file: Some(f) }
  }
}
impl Drop for TempFile {
  fn drop(&mut self) {
    //-- Se asume incorrectamente que el archivo siempre existe
    //-- Si file es None, unwrap provocará panic
    self.file.as_mut().unwrap().sync_all().unwrap();
  }
}
fn main() {
  //-- Se crea el recurso correctamente
  let mut temp = TempFile::new("ejemplo.txt");

  //-- Se rompe manualmente la invariante interna
  temp.file = None;

  //-- Se provoca un panic explícito
  //-- Durante el desenrollado se ejecutará Drop y habrá doble panic
  panic!("Error inesperado en ejecución");
}
```

Codificación recomendada

Gestión del mismo recurso asegurando que el destructor no genera `panic` y mantiene invariantes incluso si se ejecuta durante otro `panic`. Se valida el estado interno y se evita el uso de `unwrap` dentro de `Drop`, garantizando una liberación segura.

```
use std::fs::File;
use std::io::Write;
use std::thread;

struct TempFile {
  file: Option,
}
impl TempFile {
  fn new(path: &str) -> Self {
    //-- Se crea el archivo y se guarda dentro de Option
    let mut f = File::create(path).unwrap();
    writeln!(f, "datos iniciales").unwrap();
    TempFile { file: Some(f) }
  }
}
impl Drop for TempFile {
  fn drop(&mut self) {
    //-- Se toma el recurso de forma segura
    if let Some(mut f) = self.file.take() {
```

```
      //-- Se evita unwrap para no provocar panic en Drop
      if let Err(e) = f.sync_all() {
        //-- Se registra el error, pero no se interrumpe el flujo
        eprintln!("Error al sincronizar archivo: {}", e);
      }
    }
    //-- Se puede comprobar si estamos en proceso de panic
    if thread::panicking() {
      //-- No se realiza ninguna operación que pueda fallar
      eprintln!("Liberación durante panic, estado consistente");
    }
  }
}
fn main() {
  //-- Uso normal del recurso
  let _temp = TempFile::new("ejemplo_seguro.txt");

  //-- Se provoca un panic para simular fallo en ejecución
  //-- Drop no generará un segundo panic
  panic!("Error inesperado en ejecución");
}
```

2.5.5.1 Descripción

La implementación del trait `Drop` debe mantener las invariantes de los datos incluso cuando se produce un `panic` durante la ejecución del destructor. Esta regla establece que los destructores deben ser diseñados de forma que preserven la consistencia del estado del programa y no dejen recursos en estados indefinidos o corruptos.

El trait `Drop` se ejecuta automáticamente cuando un valor sale de su ámbito, pero también puede ejecutarse durante el desenrollado de la pila tras un `panic`. En estas situaciones, el destructor debe completar las operaciones críticas de limpieza sin generar panics adicionales, ya que esto resultaría en la terminación abrupta del programa.

Los riesgos asociados al incumplimiento de esta regla incluyen la **corrupción de datos**, **pérdida de recursos del sistema** y **estados inconsistentes** que pueden propagarse a otras partes del programa. Cuando un destructor falla durante el desenrollado, el programa no puede recuperarse de forma controlada.

La mitigación de estos riesgos requiere implementar destructores que sean **resistentes a fallos** y que prioricen las operaciones de limpieza más críticas. Los destructores deben evitar operaciones que puedan fallar y, cuando sea inevitable incluirlas, deben manejar los errores de forma que no comprometan la integridad del sistema. La separación de responsabilidades entre limpieza crítica y operaciones opcionales permite mantener las invariantes básicas incluso en escenarios de fallo.

Esta regla es especialmente relevante en sistemas que manejan recursos compartidos, estructuras de datos complejas o interfaces con componentes externos, donde la consistencia del estado es fundamental para el correcto funcionamiento del programa.

Riesgos asociados

- Corrupción de datos: Si `Drop` viola invariantes durante un `panic`, puede dejar el objeto en un estado inconsistente.
- Pérdida de recursos: La liberación incorrecta de recursos (archivos, memoria, sockets) puede causar fugas o accesos a recursos ya liberados.
- Condiciones de carrera: En programas concurrentes, las invariantes rotas pueden provocar interacciones inesperadas entre hilos, exponiendo datos a condiciones de carrera.
- Ejecución de código inseguro: Un `Drop` mal implementado puede ejecutar código inseguro al operar sobre datos ya corruptos.
- Segfaults o comportamientos indefinidos: Acceder a memoria ya liberada o modificada durante el `panic` puede provocar errores graves de ejecución.
- Elusión de garantías de seguridad: Romper invariantes puede hacer que otros componentes confíen erróneamente en el estado de los datos, facilitando vulnerabilidades de seguridad.
- Bloqueos o deadlocks: Finalizadores que alteran sincronización durante un `panic` podrían dejar bloqueos sin liberar, generando bloqueos en ejecución futura.

Buenas prácticas

- Evitar operaciones que puedan provocar `panic` dentro de `Drop`, gestionando errores de forma explícita para no romper invariantes.

```
//-- Drop realiza unwrap que puede provocar panic y dejar el estado inconsistente
struct Recurso {
  data: Option,
}
impl Drop for Recurso {
  fn drop(&mut self) {
    let _ = self.data.take().unwrap();
  }
}
```

```
//-- Drop maneja el Option sin provocar panic y mantiene invariantes
struct RecursoSeguro {
  data: Option,
}
impl Drop for RecursoSeguro {
```

```
  fn drop(&mut self) {
    if let Some(valor) = self.data.take() {
      drop(valor);
    }
  }
}
```

- Diseñar `Drop` para que sea idempotente y tolerante a estados parcialmente inicializados, usando banderas internas o tipos seguros.

```
//-- Drop asume que el recurso siempre está inicializado
struct Conexion {
  abierta: bool,
}
impl Drop for Conexion {
  fn drop(&mut self) {
    assert!(self.abierta);
    self.abierta = false;
  }
}
```

```
//-- Drop verifica estado y evita fallos si ocurrió un panic previo
struct ConexionSegura {
  abierta: bool,
}
impl Drop for ConexionSegura {
  fn drop(&mut self) {
    if self.abierta {
      self.abierta = false;
    }
  }
}
```

2.5.5.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 7.1**: Exception Safety.

2.5.6 [rust.err.6] - No depender de ejecución completa de destructores para mantener invariantes críticos

ID	**rust.err.6**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los destructores pueden no ejecutarse en situaciones como `panic!`, terminación abrupta del programa o ciclos de referencias. Depender exclusivamente de su ejecución para mantener **invariantes críticos** como liberación de recursos del sistema, persistencia de datos o limpieza de estado de seguridad introduce vulnerabilidades potenciales en el programa.

Codificación insegura

Gestión de un archivo de bloqueo para evitar que dos instancias de una aplicación se ejecuten al mismo tiempo. El objetivo es mostrar cómo se crea un recurso crítico y se libera en el destructor, ilustrando qué ocurre cuando el flujo del programa termina de forma abrupta antes de que los destructores se ejecuten completamente.

```
use std::fs::{File, OpenOptions};
use std::io::Write;
use std::path::Path;

struct LockFile {
  file: File,
  path: String,
}
impl LockFile {
  fn new(path: &str) -> std::io::Result<Self> {
    //-- crea el archivo de bloqueo
    let mut file = OpenOptions::new()
      .write(true)
      .create_new(true)
      .open(path)?;

    //-- escribe el PID en el archivo
    writeln!(file, "{}", std::process::id())?;

    Ok(LockFile {
      file,
      path: path.to_string(),
    })
  }
}
impl Drop for LockFile {
  fn drop(&mut self) {
```

```
    //-- elimina el archivo de bloqueo al destruir el objeto
    if Path::new(&self.path).exists() {
      let _ = std::fs::remove_file(&self.path);
    }
  }
}
fn main() {
  //-- crea el lock, asumiendo que siempre se liberará en Drop
  let _lock = LockFile::new("app.lock").expect("No se pudo crear el lock");

  //-- finaliza el proceso abruptamente
  //-- los destructores no se ejecutan y el archivo queda sin eliminar
  std::process::exit(0);
}
```

Codificación recomendada

Gestión del mismo archivo de bloqueo, pero liberando el recurso crítico de forma explícita antes de finalizar el proceso. Se busca garantizar que la invariante (no dejar un lock huérfano) no dependa exclusivamente de la ejecución automática del destructor.

```
use std::fs::{File, OpenOptions};
use std::io::Write;
use std::path::Path;

struct LockFile {
  file: File,
  path: String,
  released: bool,
}
impl LockFile {
  fn new(path: &str) -> std::io::Result<Self> {
    //-- crea el archivo de bloqueo
    let mut file = OpenOptions::new()
      .write(true)
      .create_new(true)
      .open(path)?;

    //-- escribe el PID en el archivo
    writeln!(file, "{}", std::process::id())?;

    Ok(LockFile {
      file,
      path: path.to_string(),
      released: false,
    })
  }
  fn release(&mut self) -> std::io::Result<()> {
```

```
    //-- libera explícitamente el recurso crítico
    if Path::new(&self.path).exists() {
      std::fs::remove_file(&self.path)?;
    }
    self.released = true;
    Ok(())
  }
}
impl Drop for LockFile {
  fn drop(&mut self) {
    //-- respaldo defensivo: intenta liberar si no se hizo antes
    if !self.released && Path::new(&self.path).exists() {
      let _ = std::fs::remove_file(&self.path);
    }
  }
}
fn main() {
  //-- crea el lock
  let mut lock = LockFile::new("app.lock").expect("No se pudo crear el lock");

  //-- realiza operaciones críticas
  println!("Aplicación en ejecución");

  //-- libera explícitamente antes de terminar
  lock.release().expect("No se pudo liberar el lock");

  //-- ahora se puede finalizar con seguridad
  std::process::exit(0);
}
```

2.5.6.1 Descripción

Los destructores proporcionan un mecanismo automático para liberar recursos y ejecutar código de limpieza cuando un valor sale de ámbito. Sin embargo, existen situaciones donde la ejecución de destructores puede no completarse o no ejecutarse en absoluto, comprometiendo la integridad del programa si se depende de ellos para mantener invariantes críticos del sistema.

La ejecución de destructores puede interrumpirse durante situaciones de pánico, cuando se utilizan funciones como `std::mem::forget`, o en casos de ciclos de referencias con `Rc` que impiden la liberación automática de memoria. Además, los destructores pueden fallar internamente si contienen operaciones que pueden generar errores, como escrituras a archivos o comunicaciones de red.

Esta dependencia introduce **riesgos de corrupción de estado** y **violación de invariantes de seguridad** cuando los destructores no logran ejecutar completamente las operaciones críticas esperadas. Los recursos pueden quedar en estados inconsistentes,

las estructuras de datos pueden corromperse, y las garantías de seguridad del programa pueden verse comprometidas.

La mitigación efectiva requiere **separar las operaciones críticas de los destructores** y establecer mecanismos explícitos para el mantenimiento de invariantes. Las operaciones que deben completarse para preservar la integridad del sistema necesitan ejecutarse de forma explícita y controlada, con manejo adecuado de errores y verificación de estado.

Los destructores deben limitarse a operaciones de limpieza que puedan fallar sin comprometer la seguridad general del programa. Para operaciones críticas, se recomienda implementar métodos explícitos que permitan verificar su correcta ejecución y manejar apropiadamente los casos de fallo.

Riesgos asociados

- Pérdida de invariantes críticos si el destructor no se ejecuta (por ejemplo, si ocurre un `panic` antes de finalizar el ciclo de vida del objeto).
- Posibles condiciones de carrera o deadlocks si los recursos no se liberan correctamente, dificultando la sincronización entre hilos.
- Fugas de memoria o recursos (file descriptors, sockets, locks, etc.) debido a la omisión de la liberación adecuada en situaciones no contempladas.
- Corrupción del estado interno de la aplicación o biblioteca, si se asume que el destructor siempre restaurará o actualizará recursos compartidos.
- Incremento en la superficie de ataque, facilitando exploits de denegación de servicio o escalada de privilegios por recursos retenidos inadvertidamente.
- Imposibilidad de garantizar transacciones atómicas en estructuras de datos críticas, lo que puede llevar a inconsistencias graves.

Buenas prácticas

- Validar y reforzar invariantes críticos mediante métodos explícitos en lugar de depender únicamente del destructor (`Drop`).

```
//-- Liberación implícita en Drop para restaurar invariante crítico
struct LockGuard {
  locked: bool,
}
impl Drop for LockGuard {
  fn drop(&mut self) {
    self.locked = false;
  }
}
```

```
//-- Método explícito que restablece el estado antes de finalizar el uso
struct LockGuard {
  locked: bool,
}
impl LockGuard {
  fn release(&mut self) {
    self.locked = false;
  }
}
```

- Evitar que invariantes críticos dependan de `Drop` cuando se usan tipos como `std::mem::forget` que impiden la ejecución del destructor.

```
//-- Invariante restaurado únicamente en Drop, vulnerable a mem::forget
struct Session {
  active: bool,
}
impl Drop for Session {
  fn drop(&mut self) {
    self.active = false;
  }
}
let mut s = Session { active: true };
std::mem::forget(s);
```

```
//-- Invariante gestionado mediante función explícita antes de olvidar el valor
struct Session {
  active: bool,
}
impl Session {
  fn close(&mut self) {
    self.active = false;
  }
}
let mut s = Session { active: true };
s.close();
std::mem::forget(s);
```

- Diseñar estructuras donde el estado válido sea el predeterminado y las transiciones críticas estén controladas por tipos, no por efectos secundarios en destructores.

```
//-- Cambio de estado crítico en Drop para marcar transacción como abortada
struct Transaction {
  committed: bool,
}
impl Drop for Transaction {
```

```
  fn drop(&mut self) {
    if !self.committed {
      // rollback implícito
    }
  }
}
```

```
//-- Transiciones explícitas de estado mediante API controlada
struct Transaction;
impl Transaction {
  fn commit(self) {
    // consume la transacción y confirma explícitamente
  }
  fn rollback(self) {
    // consume la transacción y revierte explícitamente
  }
}
```

2.5.6.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 6.2**: Destructors.

2.6 Concurrencia

2.6.1 [rust.sync.1] - No compartir estado mutable entre hilos sin sincronización garantizada por Send y Sync

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

El acceso concurrente a datos mutables sin sincronización adecuada genera **condiciones de carrera** y comportamiento indefinido. Los traits `Send` y `Sync` proporcionan garantías de seguridad para el intercambio de datos entre hilos, pero su uso incorrecto puede comprometer la integridad del programa.

Codificación insegura

Uso de una variable global mutable compartida entre varios hilos para incrementar un contador. La intención es mostrar cómo varios hilos modifican el mismo estado y evidenciar los riesgos de hacerlo sin un mecanismo de sincronización seguro garantizado por `Send` y `Sync`.

```
use std::thread;
use std::time::Duration;

static mut COUNTER: i32 = 0;

fn main() {
  //-- Se crean varios hilos que incrementan un contador global sin sincronización
  let mut handles = vec![];

  for _ in 0..5 {
    let handle = thread::spawn(|| {
      for _ in 0..1000 {
        //-- Acceso inseguro a estado mutable compartido
        unsafe {
          COUNTER += 1;
        }
      }
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join().unwrap();
  }
  //-- Lectura del valor final sin garantía de consistencia
  unsafe {
    println!("Valor final del contador: {}", COUNTER);
  }
  thread::sleep(Duration::from_millis(10));
}
```

Codificación recomendada

Uso de un contador compartido entre varios hilos empleando `Arc<Mutex<T>>` para garantizar sincronización segura. Se demuestra cómo compartir estado mutable entre hilos respetando las garantías de `Send` y `Sync` proporcionadas por la biblioteca estándar.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
  //-- Se crea un contador protegido por Mutex y compartido con Arc
  let counter = Arc::new(Mutex::new(0));
  let mut handles = vec![];

  for _ in 0..5 {
    let counter_clone = Arc::clone(&counter);
    let handle = thread::spawn(move || {
      for _ in 0..1000 {
```

```
        //-- Bloqueo del Mutex antes de modificar el estado compartido
        let mut num = counter_clone.lock().unwrap();
        *num += 1;
      }
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join().unwrap();
  }
  //-- Lectura segura del valor final mediante bloqueo
  let result = counter.lock().unwrap();
  println!("Valor final del contador: {}", *result);
}
```

2.6.1.1 Descripción

El acceso concurrente a datos mutables sin sincronización adecuada constituye una fuente común de errores en programación paralela. Esta regla establece que el estado mutable compartido entre hilos debe estar protegido mediante mecanismos que garanticen las propiedades `Send` y `Sync`.

Los traits `Send` y `Sync` proporcionan garantías de seguridad a nivel de tipos para el acceso concurrente. `Send` indica que un tipo puede transferirse de forma segura entre hilos, mientras que `Sync` especifica que un tipo puede ser referenciado de forma segura desde múltiples hilos simultáneamente.

La violación de esta regla puede provocar **condiciones de carrera**, donde el resultado de la ejecución depende del orden impredecible de acceso de los hilos a los datos compartidos. Estas condiciones generan comportamientos no deterministas que pueden manifestarse como corrupción de datos, estados inconsistentes o fallos intermitentes difíciles de reproducir y diagnosticar.

El sistema de tipos actúa como primera línea de defensa, rechazando en tiempo de compilación el código que intenta compartir tipos no seguros entre hilos. Sin embargo, el uso de `unsafe` puede eludir estas verificaciones, trasladando la responsabilidad de garantizar la seguridad al programador.

La mitigación efectiva requiere el uso de primitivas de sincronización apropiadas como `Mutex`, `RwLock`, `Arc` o tipos atómicos, que implementan automáticamente los traits necesarios. Estas estructuras encapsulan el estado mutable y proporcionan acceso controlado que previene las condiciones de carrera.

En contextos donde se requiere `unsafe`, la implementación manual de `Send` y `Sync` debe acompañarse de documentación detallada que justifique por qué el tipo cumple los contratos de seguridad de estos traits, incluyendo las invariantes que garantizan el acceso seguro desde múltiples hilos.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden derivar en comportamientos indeterminados o pérdida de datos.
- Corrupción de memoria por acceso concurrente a datos no protegidos.
- Fugas de datos sensibles si un hilo accede indebidamente al estado mutable de otro.
- Bloqueos (deadlocks) imprevisibles si se intentan implementar mecanismos de sincronización manualmente y de forma incorrecta.
- Pánico del programa (`panic`) y abortos en tiempo de ejecución por violación de invariantes internas de Rust.
- Elevada dificultad de depuración y análisis de fallos debido a errores concurrentes no deterministicos.
- Ataques por escalada de privilegios si un actor malicioso logra interferir con el estado mutable compartido.

Buenas prácticas

- Evitar el uso de `static mut` para compartir estado entre hilos y utilizar primitivas seguras como `Mutex` o `RwLock`.

```
//-- Uso de variable global mutable compartida sin sincronización
static mut CONTADOR: i32 = 0;
fn incrementar() {
  unsafe {
    CONTADOR += 1;
  }
}
```

```
//-- Uso de Mutex para proteger el acceso concurrente
use std::sync::Mutex;
static CONTADOR: Mutex = Mutex::new(0);
fn incrementar() {
  let mut num = CONTADOR.lock().unwrap();
  *num += 1;
}
```

- No implementar manualmente `Send` o `Sync` para tipos que contienen punteros crudos o estado no seguro sin garantizar invariantes de seguridad.

```
//-- Implementación manual de Send sin garantizar seguridad interna
struct MiTipo {
  ptr: *mut i32,
```

```
}
unsafe impl Send for MiTipo {}
```

```
//-- Encapsulamiento seguro usando Arc y Mutex para compartir entre hilos
use std::sync::{Arc, Mutex};
struct MiTipo {
  valor: Arc>,
}
```

- Utilizar `Arc` en lugar de `Rc` cuando el dato deba compartirse entre múltiples hilos.

```
//-- Uso de Rc en entorno multihilo
use std::rc::Rc;
let dato = Rc::new(5);
let copia = Rc::clone(&dato);
```

```
//-- Uso de Arc para conteo de referencias seguro entre hilos
use std::sync::Arc;
let dato = Arc::new(5);
let copia = Arc::clone(&dato);
```

- Evitar compartir estructuras con mutabilidad interior no sincronizada como `RefCell` entre hilos; preferir `Mutex` o `RwLock`.

```
//-- Uso de RefCell compartido entre hilos
use std::cell::RefCell;
use std::sync::Arc;
let dato = Arc::new(RefCell::new(0));
```

```
//-- Uso de Mutex para mutabilidad interior segura
use std::sync::{Arc, Mutex};
let dato = Arc::new(Mutex::new(0));
```

2.6.1.2 Estándares metodológicos

- **Rust-2.SEND-SYNC**: Types are Send and Sync where possible.
- **Rustonomicon 8.1**: Races.
- **Rustonomicon 8.2**: Send and Sync.

2.6.2 [rust.sync.2] - No introducir data races mediante interior mutability sin justificación formal

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

Los tipos con **interior mutability** como `RefCell`, `Cell` o `Mutex` permiten modificar datos a través de referencias inmutables, evadiendo las verificaciones del compilador. Su uso inadecuado puede introducir **data races** y condiciones de carrera que comprometen la seguridad de memoria, especialmente en contextos concurrentes donde múltiples hilos acceden simultáneamente a los mismos datos.

Codificación insegura

Uso de *interior mutability* mediante una variable global mutable compartida entre hilos sin ningún mecanismo de sincronización. El objetivo es mostrar cómo se puede forzar la mutabilidad compartida usando `static mut` y `unsafe`, permitiendo accesos concurrentes que el compilador no puede verificar, lo que abre la puerta a condiciones de carrera y comportamiento indefinido.

```
use std::thread;

static mut COUNTER: i32 = 0;

fn main() {
  //-- Se crean múltiples hilos que incrementan una variable global compartida
  let mut handles = vec![];

  for _ in 0..10 {
    handles.push(thread::spawn(|| {
      //-- Acceso a variable mutable global sin sincronización
      for _ in 0..1000 {
        unsafe {
          COUNTER += 1;
        }
      }
    }));
  }
  for handle in handles {
    handle.join().unwrap();
  }
  //-- Lectura final sin garantías de consistencia por posibles data races
  unsafe {
    println!("Resultado final: {}", COUNTER);
  }
```

```
}
```

Codificación recomendada

Uso de mutabilidad interior controlada mediante `Arc<Mutex<T>>` para compartir estado mutable entre múltiples hilos. Se demuestra cómo encapsular el acceso concurrente a un recurso compartido utilizando primitivas de sincronización seguras, evitando condiciones de carrera y garantizando exclusión mutua.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
  //-- Se encapsula el contador en Arc y Mutex para compartirlo de forma segura
  let counter = Arc::new(Mutex::new(0));
  let mut handles = vec![];

  for _ in 0..10 {
    let counter_cloned = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
      //-- Cada hilo adquiere el lock antes de modificar el valor
      for _ in 0..1000 {
        let mut num = counter_cloned.lock().unwrap();
        *num += 1;
      }
    }));
  }
  for handle in handles {
    handle.join().unwrap();
  }
  //-- Lectura final protegida por Mutex, sin riesgo de data race
  let result = counter.lock().unwrap();
  println!("Resultado final: {}", *result);
}
```

2.6.2.1 Descripción

Los data races representan una de las vulnerabilidades más complejas en programación concurrente, donde múltiples hilos acceden simultáneamente a la misma ubicación de memoria sin sincronización adecuada. El interior mutability permite modificar datos a través de referencias inmutables, creando un mecanismo que puede eludir las garantías de seguridad del sistema de tipos.

Esta regla establece que **no se debe introducir interior mutability sin una justificación formal** que demuestre la ausencia de condiciones de carrera. Los tipos como `Cell`, `RefCell`, `Mutex`, `RwLock` y `UnsafeCell` proporcionan interior mutability y requieren análisis cuidadoso antes de su implementación.

El riesgo principal surge cuando el interior mutability se combina con concurrencia sin las protecciones apropiadas. Los data races pueden provocar **comportamiento indefinido**, corrupción de datos, inconsistencias en el estado del programa y fallos intermitentes difíciles de reproducir y depurar.

La justificación formal debe incluir un análisis que demuestre que el acceso concurrente está correctamente sincronizado o que el diseño garantiza acceso exclusivo. Este análisis debe considerar todos los posibles escenarios de ejecución concurrente y verificar que las invariantes de seguridad se mantienen.

Las estrategias de mitigación incluyen el uso de primitivas de sincronización apropiadas, la limitación del alcance del interior mutability, la implementación de protocolos de acceso bien definidos y la documentación exhaustiva de las garantías de seguridad. La verificación mediante herramientas de análisis estático y testing concurrente complementa estas medidas preventivas.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden provocar resultados impredecibles y errores difíciles de depurar.
- Corrupción de memoria por accesos concurrentes no sincronizados.
- Pérdida de integridad de datos debido a modificaciones concurrentes inconsistente.
- Comportamiento indefinido (“undefined behavior”), que puede derivar en caídas inesperadas de la aplicación.
- Vulnerabilidades de seguridad que pueden ser explotadas para ejecutar código arbitrario o alterar datos sensibles.
- Violación de garantías de seguridad y concurrencia que el compilador y el sistema de tipos de Rust intentan reforzar.
- Dificultad para escalar o mantener el código, debido a la alta complejidad y el riesgo de errores sutiles.

Buenas prácticas

- Evitar el uso de `UnsafeCell` sin sincronización explícita cuando el dato puede compartirse entre hilos.

```
//-- Uso de UnsafeCell compartido entre hilos sin sincronización
use std::cell::UnsafeCell;
use std::sync::Arc;
struct Counter {
  value: UnsafeCell,
}
let counter = Arc::new(Counter { value: UnsafeCell::new(0) });
```

```
let c1 = counter.clone();
std::thread::spawn(move || {
  unsafe { *c1.value.get() += 1; }
});
```

```
//-- Uso de Mutex para garantizar exclusión mutua entre hilos
use std::sync::{Arc, Mutex};
let counter = Arc::new(Mutex::new(0u32));
let c1 = counter.clone();
std::thread::spawn(move || {
  let mut num = c1.lock().unwrap();
  *num += 1;
});
```

- No marcar manualmente un tipo como `Sync` o `Send` cuando contiene interior mutability no sincronizada.

```
//-- Implementación manual de Sync sin garantizar sincronización interna
use std::cell::RefCell;
struct SharedData {
  data: RefCell,
}
unsafe impl Sync for SharedData {}
```

```
//-- Eliminación de implementación manual y uso de Mutex para seguridad en
concurrencia
use std::sync::Mutex;
struct SharedData {
  data: Mutex,
}
```

- Restringir `RefCell` a contextos monohilo y evitar compartirlo mediante `Arc`.

```
//-- Compartición de RefCell entre hilos usando Arc
use std::cell::RefCell;
use std::sync::Arc;
let value = Arc::new(RefCell::new(5));
let v1 = value.clone();
std::thread::spawn(move || {
  *v1.borrow_mut() += 1;
});
```

```
//-- Uso de AtomicUsize para mutación segura sin bloqueo
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
let value = Arc::new(AtomicUsize::new(5));
```

```
let v1 = value.clone();
std::thread::spawn(move || {
  v1.fetch_add(1, Ordering::SeqCst);
});
```

- Documentar y encapsular formalmente cualquier uso de `unsafe` relacionado con interior mutability.

```
//-- Bloque unsafe expuesto directamente al consumidor del API
use std::cell::UnsafeCell;
pub struct Buffer {
  data: UnsafeCell>,
}
pub fn push(&self, value: u8) {
  unsafe { (*self.data.get()).push(value); }
}
```

```
//-- Encapsulamiento con sincronización interna y sin exponer unsafe
use std::sync::Mutex;
pub struct Buffer {
  data: Mutex>,
}
pub fn push(&self, value: u8) {
  let mut vec = self.data.lock().unwrap();
  vec.push(value);
}
```

2.6.2.2 Estándares metodológicos

- **Rust-2.SEND-SYNC**: Types are Send and Sync where possible.
- **Rustonomicon 8.1**: Races.
- **Rustonomicon 8.3**: Atomics.

2.6.3 [rust.sync.3] - No asumir orden de ejecución ni visibilidad de memoria sin primitivas de sincronización

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

Los accesos concurrentes a memoria compartida sin mecanismos de sincronización explícitos pueden generar **condiciones de carrera** y comportamientos indefinidos. El compilador y el procesador pueden reordenar operaciones de memoria por optimización, haciendo que el orden de ejecución observado difiera del orden del código

fuente. Sin `Mutex`, `RwLock`, operaciones atómicas o barreras de memoria, no existe garantía sobre cuándo serán visibles las modificaciones entre hilos diferentes.

Codificación insegura

Acceso concurrente a un contador compartido entre hilos sin utilizar primitivas de sincronización. Se busca mostrar cómo la falta de garantías de orden de ejecución y visibilidad de memoria puede producir resultados inconsistentes y condiciones de carrera.

```
use std::thread;
use std::time::Duration;

//-- Contador global mutable compartido sin sincronización
static mut COUNTER: u32 = 0;

fn main() {
  //-- Vector para almacenar los manejadores de los hilos
  let mut handles = vec![];

  for _ in 0..10 {
    //-- Se crean 10 hilos que incrementan el contador
    let handle = thread::spawn(|| {
      for _ in 0..1000 {
        //-- Acceso a memoria compartida sin sincronización (condición de carrera)
        unsafe {
          COUNTER += 1;
        }
      }
    });
    handles.push(handle);
  }
  //-- Espera a que todos los hilos finalicen
  for handle in handles {
    handle.join().unwrap();
  }
  //-- El resultado puede ser menor a 10000 debido a la falta de sincronización
  unsafe {
    println!("Valor final del contador: {}", COUNTER);
  }
}
```

Codificación recomendada

Incremento concurrente de un contador compartido utilizando primitivas de sincronización que garantizan exclusión mutua y visibilidad de memoria entre hilos. Se pretende mostrar cómo asegurar consistencia y orden correcto en entornos multihilo.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
  //-- Contador protegido por Mutex para garantizar exclusión mutua
  let counter = Arc::new(Mutex::new(0u32));
  let mut handles = vec![];

  for _ in 0..10 {
    //-- Se clona la referencia compartida para cada hilo
    let counter_cloned = Arc::clone(&counter);
    let handle = thread::spawn(move || {
      for _ in 0..1000 {
        //-- Se adquiere el bloqueo antes de modificar el valor
        let mut num = counter_cloned.lock().unwrap();
        *num += 1;
      }
    });
    handles.push(handle);
  }
  //-- Espera a que todos los hilos finalicen
  for handle in handles {
    handle.join().unwrap();
  }
  //-- Acceso seguro al valor final con garantía de visibilidad de memoria
  let result = counter.lock().unwrap();
  println!("Valor final del contador: {}", *result);
}
```

2.6.3.1 Descripción

La programación concurrente introduce complejidad en el orden de ejecución de las operaciones y en la visibilidad de los cambios de memoria entre hilos. Sin las primitivas de sincronización adecuadas, no existe garantía sobre cuándo se ejecutarán las operaciones ni sobre cuándo los cambios realizados por un hilo serán visibles para otros hilos.

El compilador y el procesador pueden reordenar las instrucciones para optimizar el rendimiento, alterando el orden aparente de ejecución respecto al código fuente. Además, cada hilo puede mantener copias locales de los datos en cachés o registros, lo que significa que las modificaciones realizadas por un hilo pueden no ser inmediatamente visibles para otros hilos.

Esta regla establece que **no se debe asumir un orden específico de ejecución** entre operaciones concurrentes ni que los cambios de memoria sean inmediatamente visibles entre hilos sin utilizar mecanismos de sincronización explícitos. Las operaciones de

lectura y escritura sobre datos compartidos requieren coordinación mediante primitivas como `Mutex`, `RwLock`, canales, o tipos atómicos.

Los riesgos asociados incluyen condiciones de carrera, lecturas de datos inconsistentes, corrupción de estructuras de datos compartidas y comportamiento no determinista del programa. Estos problemas pueden manifestarse de forma intermitente y ser difíciles de reproducir y depurar.

La mitigación se basa en el uso sistemático de primitivas de sincronización apropiadas para cada caso de uso. Los tipos atómicos proporcionan operaciones con garantías de ordenamiento específicas, mientras que los mutex y locks ofrecen exclusión mutua para secciones críticas. Los canales permiten comunicación segura entre hilos sin compartir memoria directamente.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden provocar resultados impredecibles o corrupción de datos.
- Lectura o escritura de datos inconsistentes, afectando la integridad de la información compartida entre hilos.
- Actualización no atómica de variables compartidas, vulnerando la coherencia entre los hilos.
- Exposición a ataques de datos temporales, permitiendo que actores maliciosos exploten estados intermedios no protegidos.
- Difícil detección y depuración de errores, aumentando el costo de mantenimiento y la complejidad del software.
- Bugs dependientes de la plataforma o el hardware (heisenbugs), que pueden no reproducirse consistentemente entre ejecuciones.
- Violación de garantías de seguridad de Rust, anulando la promesa de seguridad de memoria y de concurrencia del lenguaje.
- Posibilidad de que datos sensibles queden expuestos temporalmente en memoria compartida.
- Degradación del rendimiento por bucles de espera activa o fallos de coherencia en caches de CPU.

Buenas prácticas

- Utilizar primitivas de sincronización (`Mutex`, `RwLock`) para proteger el acceso concurrente a datos compartidos.

```
//-- acceso concurrente a variable estática sin sincronización
static mut CONTADOR: u32 = 0;
```

```
fn incrementar() {
  unsafe { CONTADOR += 1; }
}
```

```
//-- acceso protegido con Mutex para garantizar exclusión mutua
use std::sync::Mutex;
static CONTADOR_SEGURO: Mutex = Mutex::new(0);
fn incrementar() {
  let mut valor = CONTADOR_SEGURO.lock().unwrap();
  *valor += 1;
}
```

- Usar tipos atómicos cuando se requieran operaciones simples compartidas entre hilos.

```
//-- incremento sin atomicidad en entorno multihilo
use std::thread;
let mut contador = 0;
thread::spawn(|| {
  contador += 1;
});
```

```
//-- uso de AtomicUsize con orden de memoria explícito
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
let contador = Arc::new(AtomicUsize::new(0));
let c = Arc::clone(&contador);
thread::spawn(move || {
  c.fetch_add(1, Ordering::SeqCst);
});
```

- Compartir datos entre hilos mediante `Arc` en lugar de asumir copias o referencias seguras.

```
//-- intento de compartir referencia mutable entre hilos
let mut datos = vec![1, 2, 3];
std::thread::spawn(|| {
  datos.push(4);
});
```

```
//-- uso de Arc y Mutex para compartir y modificar datos
use std::sync::{Arc, Mutex};
let datos = Arc::new(Mutex::new(vec![1, 2, 3]));
let datos_clon = Arc::clone(&datos);
std::thread::spawn(move || {
  let mut v = datos_clon.lock().unwrap();
```

```
  v.push(4);
});
```

- Especificar explícitamente el orden de memoria en operaciones atómicas y no asumir consistencia implícita.

```
//-- carga atómica con orden relajado sin considerar visibilidad
use std::sync::atomic::{AtomicBool, Ordering};
static LISTO: AtomicBool = AtomicBool::new(false);
if LISTO.load(Ordering::Relaxed) {
  procesar();
}
```

```
//-- uso de Acquire/Release para garantizar sincronización adecuada
if LISTO.load(Ordering::Acquire) {
  procesar();
}
LISTO.store(true, Ordering::Release);
```

2.6.3.2 Estándares metodológicos

- **Rust-2.SEND-SYNC**: Types are Send and Sync where possible.
- **Rustonomicon 8.1**: Races.
- **Rustonomicon 8.3**: Atomics.

2.6.4 [rust.sync.4] - No usar ordenamientos Relaxed sin justificación formal basada en el modelo de memoria

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

Los ordenamientos de memoria `Relaxed` permiten reordenamientos extensivos de operaciones atómicas, lo que puede generar **condiciones de carrera sutiles** y comportamientos impredecibles en código concurrente. Su uso sin una comprensión profunda del modelo de memoria puede introducir errores difíciles de detectar y reproducir.

Codificación insegura

Se muestra un escenario con dos hilos donde uno escribe un dato compartido y luego activa una bandera atómica para indicar que el dato está listo. El objetivo es entender

cómo el uso de ordenamientos débiles puede permitir reordenamientos que rompan la garantía de visibilidad entre hilos.

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;
use std::time::Duration;

static FLAG: AtomicBool = AtomicBool::new(false);
static mut DATA: u32 = 0;

fn main() {
  //-- Hilo productor que escribe el dato y activa la bandera
  let producer = thread::spawn(|| {
    unsafe {
      //-- Escritura del dato compartido
      DATA = 42;
    }
    //-- Uso de Ordering::Relaxed sin justificación formal
    FLAG.store(true, Ordering::Relaxed);
  });

  //-- Hilo consumidor que espera la bandera
  let consumer = thread::spawn(|| {
    while !FLAG.load(Ordering::Relaxed) {
      //-- Espera activa
      thread::sleep(Duration::from_millis(1));
    }
    unsafe {
      //-- Lectura del dato compartido
      println!("Valor leído: {}", DATA);
    }
  });

  producer.join().unwrap();
  consumer.join().unwrap();
}
```

Codificación recomendada

Se presenta el mismo escenario, pero utilizando un ordenamiento que establece una relación de sincronización clara entre hilos. El objetivo es garantizar que la escritura del dato ocurra antes de que el otro hilo observe la bandera activada, evitando reordenamientos no deseados.

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;
use std::time::Duration;
```

```
static FLAG: AtomicBool = AtomicBool::new(false);
static mut DATA: u32 = 0;

fn main() {
  //-- Hilo productor que escribe el dato y activa la bandera
  let producer = thread::spawn(|| {
    unsafe {
      //-- Escritura del dato compartido
      DATA = 42;
    }
    //-- Uso de Ordering::Release para publicar el dato correctamente
    FLAG.store(true, Ordering::Release);
  });

  //-- Hilo consumidor que espera la bandera
  let consumer = thread::spawn(|| {
    while !FLAG.load(Ordering::Acquire) {
      //-- Espera activa
      thread::sleep(Duration::from_millis(1));
    }
    unsafe {
      //-- Lectura del dato compartido con garantía de visibilidad
      println!("Valor leído: {}", DATA);
    }
  });

  producer.join().unwrap();
  consumer.join().unwrap();
}
```

2.6.4.1 Descripción

Los ordenamientos `Relaxed` en operaciones atómicas proporcionan las garantías de sincronización más débiles del modelo de memoria. Estas operaciones garantizan únicamente la atomicidad de la operación individual, sin establecer relaciones de orden con otras operaciones de memoria ni crear puntos de sincronización entre hilos.

El uso indiscriminado de ordenamientos `Relaxed` puede introducir **condiciones de carrera sutiles** y comportamientos no deterministas que son extremadamente difíciles de detectar durante las pruebas. Estos problemas suelen manifestarse de forma intermitente y pueden variar según la arquitectura del procesador, las optimizaciones del compilador y las condiciones de carga del sistema.

La ausencia de garantías de orden puede provocar que las operaciones se ejecuten en secuencias inesperadas, causando **inconsistencias en el estado compartido** entre hilos. Los efectos pueden incluir lecturas de valores obsoletos, escrituras que no se propagan correctamente entre núcleos del procesador, y violaciones de invariantes que dependen de un orden específico de operaciones.

La mitigación de estos riesgos requiere un **análisis formal del modelo de memoria** que demuestre que el ordenamiento relajado es suficiente para el caso de uso específico. Este análisis debe considerar todas las posibles reordenaciones de operaciones y verificar que ninguna secuencia de ejecución válida pueda violar las propiedades de seguridad del programa.

Como alternativa, se pueden emplear ordenamientos más estrictos como `Acquire`, `Release` o `SeqCst` que proporcionan garantías de sincronización más fuertes. Aunque estos ordenamientos pueden tener un costo de rendimiento superior, ofrecen mayor predictibilidad y facilitan el razonamiento sobre la corrección del programa concurrente.

Riesgos asociados

- Condiciones de carrera impredecibles debido a supuestos incorrectos sobre la visibilidad de las operaciones entre hilos.
- Violaciones de integridad de datos por inconsistencias causadas por escrituras y lecturas atómicas con semántica `Relaxed`.
- Difíciles fallos de depuración y errores sutiles que pueden no reproducirse consistentemente (heisenbugs).
- Saltos en la ejecución que pueden omitir sincronización necesaria, derivando en corrupción de memoria.
- Posibilidad de romper invariantes lógicos internos de la aplicación, generando comportamientos no deseados o inseguros.
- Exposición accidental de información sensible si las operaciones concurrentes no se coordinan como se espera.
- Ataques de escalada de privilegios o denegación de servicio por manipulación de condiciones de carrera.
- Dificultad para verificar formalmente la corrección del programa debido a la falta de garantías de orden.

Buenas prácticas

- Utilizar ordenamientos `Acquire` / `Release` cuando exista sincronización entre hilos en lugar de emplear `Relaxed` sin justificación formal.

```
//-- Uso de ordenamiento Relaxed en señalización entre hilos
static FLAG: AtomicBool = AtomicBool::new(false);
fn writer() {
  DATA.store(42, Ordering::Relaxed);
  FLAG.store(true, Ordering::Relaxed);
}
fn reader() {
  if FLAG.load(Ordering::Relaxed) {
```

```
    println!("{}", DATA.load(Ordering::Relaxed));
  }
}
```

```
//-- Uso de Release/Acquire para garantizar visibilidad de memoria
static FLAG: AtomicBool = AtomicBool::new(false);
fn writer() {
  DATA.store(42, Ordering::Relaxed);
  FLAG.store(true, Ordering::Release);
}
fn reader() {
  if FLAG.load(Ordering::Acquire) {
    println!("{}", DATA.load(Ordering::Relaxed));
  }
}
```

- Preferir `Ordering::SeqCst` cuando no se haya realizado un análisis formal del modelo de memoria que justifique una optimización más débil.

```
//-- Uso de Relaxed en contador compartido sin análisis formal
static COUNTER: AtomicUsize = AtomicUsize::new(0);
fn increment() {
  COUNTER.fetch_add(1, Ordering::Relaxed);
}
```

```
//-- Uso de SeqCst para mantener un orden global consistente
static COUNTER: AtomicUsize = AtomicUsize::new(0);
fn increment() {
  COUNTER.fetch_add(1, Ordering::SeqCst);
}
```

- Evitar combinar múltiples accesos `Relaxed` que dependan entre sí; usar barreras explícitas o primitivas de sincronización seguras.

```
//-- Múltiples cargas Relaxed con dependencia implícita
static READY: AtomicBool = AtomicBool::new(false);
static VALUE: AtomicUsize = AtomicUsize::new(0);
fn read_data() {
  if READY.load(Ordering::Relaxed) {
    let v = VALUE.load(Ordering::Relaxed);
    println!("{}", v);
  }
}
```

```
//-- Uso de Acquire para asegurar consistencia entre bandera y dato
static READY: AtomicBool = AtomicBool::new(false);
```

```
static VALUE: AtomicUsize = AtomicUsize::new(0);
fn read_data() {
  if READY.load(Ordering::Acquire) {
    let v = VALUE.load(Ordering::Relaxed);
    println!("{}", v);
  }
}
```

2.6.4.2 Estándares metodológicos

- **Rust-2.SEND-SYNC**: Types are Send and Sync where possible.
- **Rustonomicon 8.3**: Atomics.

2.6.5 [rust.sync.5] - No mezclar atomics y locks sin un modelo de sincronización coherente

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

La combinación inadecuada de operaciones atómicas y mecanismos de bloqueo puede generar **condiciones de carrera e inconsistencias de memoria** en programas concurrentes. Sin un modelo de sincronización coherente, el acceso simultáneo a datos compartidos mediante `Atomic` y `Mutex` produce comportamientos impredecibles y errores difíciles de detectar.

Codificación insegura

Se muestra un escenario con múltiples hilos donde se intenta coordinar el acceso a un recurso compartido usando un `AtomicBool` junto con un `Mutex`, pero sin un modelo de sincronización claro. La intención es controlar si un dato está listo para usarse. El objetivo es observar cómo la mezcla de primitivas sin una estrategia coherente puede generar condiciones de carrera lógicas y estados inconsistentes.

```
use std::sync::{Arc, Mutex};
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
  //-- Bandera atómica para indicar si el dato está listo
  let ready = Arc::new(AtomicBool::new(false));
  //-- Recurso protegido con Mutex
  let data = Arc::new(Mutex::new(0));

  let ready_writer = Arc::clone(&ready);
```

```
  let data_writer = Arc::clone(&data);

  let writer = thread::spawn(move || {
    //-- Se adquiere el lock y se modifica el dato
    let mut value = data_writer.lock().unwrap();
    *value = 42;
    //-- Se marca como listo usando atomic sin coordinación clara
    ready_writer.store(true, Ordering::Relaxed);
  });

  let ready_reader = Arc::clone(&ready);
  let data_reader = Arc::clone(&data);

  let reader = thread::spawn(move || {
    //-- Se consulta la bandera fuera del lock
    if ready_reader.load(Ordering::Relaxed) {
      //-- Se asume que el dato ya está listo
      let value = data_reader.lock().unwrap();
      println!("Valor leído: {}", *value);
    } else {
      println!("Dato no listo");
    }
  });

  writer.join().unwrap();
  reader.join().unwrap();
}
```

Codificación recomendada

Se presenta el mismo escenario de coordinación entre hilos, pero utilizando un único mecanismo de sincronización coherente. Tanto el dato como su estado se agrupan dentro de una estructura protegida por `Mutex`, evitando mezclar atomics y locks. El objetivo es garantizar consistencia y claridad en el modelo de concurrencia.

```
use std::sync::{Arc, Mutex};
use std::thread;

struct SharedState {
  //-- Indica si el dato está listo
  ready: bool,
  //-- Dato compartido
  value: i32,
}
fn main() {
  //-- Estado compartido protegido únicamente por Mutex
  let state = Arc::new(Mutex::new(SharedState {
    ready: false,
```

```
    value: 0,
  }));

  let state_writer = Arc::clone(&state);

  let writer = thread::spawn(move || {
    //-- Se adquiere el lock y se actualiza todo el estado de forma coherente
    let mut shared = state_writer.lock().unwrap();
    shared.value = 42;
    shared.ready = true;
  });

  let state_reader = Arc::clone(&state);

  let reader = thread::spawn(move || {
    //-- Se adquiere el lock antes de comprobar el estado
    let shared = state_reader.lock().unwrap();
    if shared.ready {
      println!("Valor leído: {}", shared.value);
    } else {
      println!("Dato no listo");
    }
  });

  writer.join().unwrap();
  reader.join().unwrap();
}
```

2.6.5.1 Descripción

La combinación de operaciones atómicas y mecanismos de bloqueo tradicionales requiere un diseño de sincronización coherente y bien definido. Cuando se utilizan ambos enfoques simultáneamente sin una estrategia clara, se pueden introducir condiciones de carrera sutiles, inconsistencias en el estado compartido y comportamientos impredecibles en entornos concurrentes.

Las operaciones atómicas proporcionan garantías de **consistencia a nivel de hardware** para operaciones individuales, mientras que los locks ofrecen **exclusión mutua** para secciones críticas más amplias. La mezcla indiscriminada de estos mecanismos puede crear situaciones donde las garantías de uno interfieren con las del otro, especialmente cuando diferentes hilos acceden a los mismos datos usando estrategias de sincronización distintas.

Los riesgos principales incluyen la creación de **deadlocks** cuando se combinan múltiples primitivas de sincronización, la introducción de **condiciones de carrera** debido a suposiciones incorrectas sobre el orden de las operaciones, y la generación de

inconsistencias en la memoria cuando las barreras de memoria de los atomics no se alinean correctamente con las garantías de los locks.

La mitigación efectiva requiere establecer un **modelo de sincronización uniforme** para cada estructura de datos compartida. Esto implica documentar claramente qué mecanismo de sincronización protege cada recurso y mantener consistencia en el acceso a través de toda la aplicación. Cuando sea necesario combinar ambos enfoques, se debe diseñar una jerarquía clara de sincronización que evite dependencias circulares y garantice un orden determinista de adquisición de recursos.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden causar corrupción de memoria o datos inconsistentes.
- Interbloqueos (deadlocks) por bloqueos cruzados y no planificados entre locks y operaciones atómicas.
- Bloqueos activos (livelocks) por intentos repetidos fallidos de adquisición de locks mientras se usan primitivas atómicas.
- Comportamientos no deterministas difíciles de depurar debido a la interleaving impredecible entre locks y atomics.
- Fugas de información o violaciones de invariantes por saltarse garantías de memoria proporcionadas por un solo mecanismo de sincronización.
- Dependencia de la semántica específica de la plataforma, lo que causa problemas al portar el código entre arquitecturas.
- Aumento de la superficie de ataque para exploits como TOCTOU (Time-Of-Check to Time-Of-Use).
- Violaciones al modelo de memoria de Rust, potencialmente llevando a errores de seguridad que `unsafe` no puede prevenir ni detectar fácilmente.

Buenas prácticas

- Definir un único modelo de sincronización (solo atomics o solo locks) para proteger el mismo recurso compartido.

```
//-- Uso combinado de AtomicBool y Mutex para el mismo estado compartido
static FLAG: AtomicBool = AtomicBool::new(false);
static DATA: Mutex = Mutex::new(0);
fn update() {
  if !FLAG.load(Ordering::Relaxed) {
    let mut d = DATA.lock().unwrap();
    *d += 1;
    FLAG.store(true, Ordering::Relaxed);
  }
```

```
}
```

```
//-- Uso exclusivo de Mutex para proteger estado y condición
static DATA: Mutex<(bool, u32)> = Mutex::new((false, 0));
fn update() {
  let mut data = DATA.lock().unwrap();
  if !data.0 {
    data.1 += 1;
    data.0 = true;
  }
}
```

- Si se requieren atomics, utilizar órdenes de memoria coherentes y evitar proteger parcialmente el mismo recurso con locks.

```
//-- Atomic con Ordering::Relaxed y acceso concurrente adicional con Mutex
static READY: AtomicBool = AtomicBool::new(false);
static VALUE: Mutex = Mutex::new(0);
fn writer() {
  let mut v = VALUE.lock().unwrap();
  *v = 42;
  READY.store(true, Ordering::Relaxed);
}
fn reader() {
  if READY.load(Ordering::Relaxed) {
    let v = VALUE.lock().unwrap();
    println!("{}", *v);
  }
}
```

```
//-- Uso exclusivo de atomics con orden Acquire/Release para publicación segura
static VALUE: AtomicU32 = AtomicU32::new(0);
static READY: AtomicBool = AtomicBool::new(false);
fn writer() {
  VALUE.store(42, Ordering::Relaxed);
  READY.store(true, Ordering::Release);
}
fn reader() {
  if READY.load(Ordering::Acquire) {
    let v = VALUE.load(Ordering::Relaxed);
    println!("{}", v);
  }
}
```

- Evitar usar atomics como mecanismo auxiliar de verificación externa cuando el recurso ya está completamente protegido por un lock.

```
//-- Verificación previa con AtomicUsize antes de adquirir el lock
static COUNT: AtomicUsize = AtomicUsize::new(0);
static ITEMS: Mutex> = Mutex::new(Vec::new());
fn add_item(x: u32) {
  if COUNT.load(Ordering::Relaxed) < 10 {
    let mut items = ITEMS.lock().unwrap();
    items.push(x);
    COUNT.fetch_add(1, Ordering::Relaxed);
  }
}
```

```
//-- Control completo del límite dentro del Mutex
static ITEMS: Mutex> = Mutex::new(Vec::new());
fn add_item(x: u32) {
  let mut items = ITEMS.lock().unwrap();
  if items.len() < 10 {
    items.push(x);
  }
}
```

2.6.5.2 Estándares metodológicos

- **Rust-2.SEND-SYNC**: Types are Send and Sync where possible.
- **Rustonomicon 8.3**: Atomics.

2.6.6 [rust.sync.6] - No asumir ejecución completa de futures ante cancelación

ID	rust.sync.6	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Los futures pueden ser cancelados en cualquier momento durante su ejecución, interrumpiendo el flujo normal del programa. Esta cancelación abrupta puede ocurrir cuando se abandona un future, se agota un timeout o se produce una terminación anticipada del runtime. El problema surge cuando el código asume que todas las operaciones asíncronas se completarán exitosamente, sin considerar que la **cancelación puede dejar recursos sin liberar** o estados inconsistentes en el sistema.

Codificación insegura

Se muestra un programa asíncrono donde una tarea realiza una operación crítica (actualizar un estado compartido) y se asume que siempre terminará su ejecución. La intención es observar qué ocurre cuando la future es cancelada antes de completar su lógica, dejando el sistema en un estado inconsistente por no contemplar la cancelación.

```
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, Duration};

//-- Simula una operación crítica que tarda en completarse
async fn operacion_critica(estado: Arc<Mutex<bool>>) {
  //-- Inicio de la operación
  println!("Iniciando operación crítica...");

  //-- Simula trabajo prolongado
  sleep(Duration::from_secs(5)).await;

  //-- Se marca el estado como completado
  let mut flag = estado.lock().unwrap();
  *flag = true;
  println!("Operación completada.");
}
#[tokio::main]
async fn main() {
  let estado = Arc::new(Mutex::new(false));

  //-- Se lanza la tarea asíncrona
  let tarea = tokio::spawn(operacion_critica(estado.clone()));

  //-- Se espera menos tiempo del necesario y se cancela
  sleep(Duration::from_secs(1)).await;
  drop(tarea);

  //-- Se asume incorrectamente que la operación terminó
  let flag = estado.lock().unwrap();
  println!("¿Estado completado?: {}", *flag);
}
```

Codificación recomendada

Se presenta un programa asíncrono que contempla explícitamente la posible cancelación de una future. La lógica crítica se ejecuta en una sección que garantiza consistencia del estado incluso si la tarea es interrumpida. Se aprende a no depender de la ejecución completa de la future y a diseñar el flujo para manejar cancelaciones de forma segura.

```
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, timeout, Duration};

//-- Simula una operación crítica segura ante cancelación
async fn operacion_critica_segura(estado: Arc<Mutex<bool>>) -> Result<(), ()> {
  println!("Iniciando operación crítica segura...");

  //-- Trabajo dividido en pasos pequeños verificables
```

```
  sleep(Duration::from_secs(2)).await;

  //-- Actualización protegida del estado
  {
    let mut flag = estado.lock().unwrap();
    *flag = true;
  }
  println!("Operación completada correctamente.");
  Ok(())
}
#[tokio::main]
async fn main() {
  let estado = Arc::new(Mutex::new(false));

  //-- Se ejecuta con un límite de tiempo controlado
  let resultado = timeout(
    Duration::from_secs(1),
    operacion_critica_segura(estado.clone())
  ).await;

  match resultado {
    Ok(Ok(())) => println!("La operación finalizó con éxito."),
    Ok(Err(_)) => println!("La operación devolvió error."),
    Err(_) => {
      //-- Manejo explícito de cancelación por timeout
      println!("La operación fue cancelada por timeout.");
    }
  }
  //-- Se verifica el estado real después de la posible cancelación
  let flag = estado.lock().unwrap();
  println!("¿Estado completado?: {}", *flag);
}
```

2.6.6.1 Descripción

Los futures pueden ser cancelados en cualquier momento durante su ejecución, interrumpiendo el flujo normal del programa antes de que alcancen su estado de finalización. Esta característica del modelo asíncrono introduce complejidades específicas en el manejo de recursos y la consistencia del estado.

La cancelación ocurre cuando se abandona un future sin completar su ejecución, típicamente mediante operaciones como `Drop` del future o cuando se selecciona otra rama en operaciones concurrentes. El sistema de ejecución asíncrona no garantiza que el future llegue a ejecutar todas sus instrucciones programadas.

Esta interrupción abrupta puede generar **inconsistencias en el estado de la aplicación**, especialmente cuando el future maneja recursos compartidos, realiza operaciones de

escritura parciales o mantiene invariantes que dependen de la ejecución completa de una secuencia de operaciones.

Los riesgos se manifiestan principalmente en **fugas de recursos**, estados corruptos en estructuras de datos compartidas, y violaciones de invariantes del programa. Las operaciones que requieren atomicidad o que modifican múltiples elementos relacionados son particularmente vulnerables a estos problemas.

La mitigación efectiva requiere diseñar futures que sean **seguros ante cancelación**, implementando mecanismos de limpieza automática y evitando asumir que las operaciones asíncronas completarán su ejecución. Las técnicas incluyen el uso de patrones RAII, operaciones atómicas, y estructuración del código que minimice las ventanas críticas donde la cancelación puede causar inconsistencias.

La gestión adecuada de este aspecto es especialmente importante en sistemas que manejan recursos externos, realizan operaciones de red, o mantienen estado compartido entre múltiples tareas concurrentes.

Riesgos asociados

- Condición de carrera por acceso inconsistente a recursos compartidos.
- Pérdida de datos intermedios o estados importantes al no finalizar operaciones críticas.
- Liberación incompleta o incorrecta de recursos (sockets, archivos abiertos, locks).
- Corrupción de estado interno si el future modifica estructuras y es cancelado a la mitad.
- Inconsistencias y errores lógicos en flujos de trabajo transaccionales.
- Incremento de vulnerabilidad ante ataques DoS por acumulación de recursos no liberados.
- Escape de información sensible si no se limpian estados o buffers a tiempo.
- Deadlocks si otros futures esperan una señalización nunca enviada.
- Exposición accidental de datos parciales en respuestas al usuario o por logs incompletos.
- Desincronización en sistemas distribuidos debido a operaciones a medio terminar.

Buenas prácticas

- No asumir que un `Future` completará su ejecución; diseñar operaciones críticas para que sean atómicas o resistentes a cancelaciones.

```
//-- Operación en varios pasos sin garantizar consistencia ante cancelación
async fn actualizar_estado(db: &Database) {
  db.marcar_como_procesando().await;
  db.guardar_resultado().await;
}
```

```
//-- Operación atómica encapsulada en una transacción resistente a cancelación
async fn actualizar_estado(db: &Database) -> Result<(), Error> {
  let mut tx = db.iniciar_transaccion().await?;
  tx.marcar_como_procesando().await?;
  tx.guardar_resultado().await?;
  tx.confirmar().await?;
  Ok(())
}
```

- Implementar mecanismos de limpieza o compensación para recursos críticos cuando un `Future` pueda ser cancelado.

```
//-- Recurso adquirido sin manejo ante cancelación
async fn procesar_archivo() {
  let mut archivo = abrir_archivo().await;
  archivo.escribir_datos().await;
}
```

```
//-- Recurso protegido con patrón RAII para liberar en Drop
struct ArchivoSeguro {
  inner: Archivo,
}
impl Drop for ArchivoSeguro {
  fn drop(&mut self) {
    self.inner.cerrar_seguro();
  }
}
async fn procesar_archivo() {
  let mut archivo = ArchivoSeguro { inner: abrir_archivo().await };
  archivo.inner.escribir_datos().await;
}
```

- Evitar depender de efectos secundarios posteriores a un `.await` para mantener invariantes de seguridad.

```
//-- Cambio de estado antes de await que puede quedar inconsistente
async fn cambiar_bandera(obj: &mut Objeto) {
  obj.activo = false;
  operacion_larga().await;
  obj.activo = true;
```

```
}
```

```
//-- Estado restaurado mediante guardia que asegura consistencia
struct GuardiaEstado<'a> {
  obj: &'a mut Objeto,
}
impl<'a> Drop for GuardiaEstado<'a> {
  fn drop(&mut self) {
    self.obj.activo = true;
  }
}
async fn cambiar_bandera(obj: &mut Objeto) {
  obj.activo = false;
  let _guardia = GuardiaEstado { obj };
  operacion_larga().await;
}
```

- Diseñar tareas asincrónicas críticas para que sean idempotentes, permitiendo reintentos seguros tras cancelación.

```
//-- Inserción que puede duplicar datos tras reintento
async fn registrar_evento(db: &Database, evento: Evento) {
  db.insertar(evento).await;
}
```

```
//-- Inserción idempotente validando existencia previa
async fn registrar_evento(db: &Database, evento: Evento) -> Result<(), Error> {
  if !db.existe(&evento.id).await? {
    db.insertar(evento).await?;
  }
  Ok(())
}
```

2.6.6.2 Estándares metodológicos

- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 7.1**: Exception Safety.
- **Rustonomicon 7.2**: Poisoning.

2.6.7 [rust.sync.7] - No mantener invariantes críticas a través de puntos await

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

Los puntos `await` representan interrupciones en la ejecución asíncrona donde el control puede transferirse a otras tareas. Durante estas suspensiones, el estado compartido puede modificarse por código concurrente, violando las **invariantes críticas** que el programador asume válidas antes y después del punto de suspensión. Esta situación genera condiciones de carrera sutiles y errores de concurrencia difíciles de detectar.

Codificación insegura

Gestión de un estado compartido donde se modifica una estructura protegida por un `Mutex` y se realiza una operación asíncrona mientras el estado está en una condición intermedia. Se busca observar qué ocurre cuando una invariante crítica (por ejemplo, si está marcado como procesando, los datos deben ser consistentes) se mantiene abierta durante un punto `await`, lo que puede provocar bloqueos o estados inconsistentes.

```
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, Duration};

struct SharedState {
  processing: bool,
  value: i32,
}
#[tokio::main]
async fn main() {
  //-- Inicializa el estado compartido
  let state = Arc::new(Mutex::new(SharedState { processing: false, value: 0 }));

  //-- Clona la referencia para simular concurrencia
  let state_clone = Arc::clone(&state);

  let handle = tokio::spawn(async move {
    //-- Adquiere el lock y modifica el estado
    let mut data = state_clone.lock().unwrap();
    data.processing = true;
    data.value = 42;

    //-- Se mantiene el lock y la invariante incompleta durante el await
    sleep(Duration::from_secs(2)).await;

    //-- Finaliza la operación y restaura la invariante
    data.processing = false;
  });
```

```
  //-- Espera a que termine la tarea
  handle.await.unwrap();

  //-- Verifica el estado final
  let final_state = state.lock().unwrap();
  println!("Processing: {}, Value: {}", final_state.processing, final_state.value);
}
```

Codificación recomendada

Gestión del mismo estado compartido, asegurando que la invariante crítica se restaura antes de alcanzar cualquier punto `await`. Se limita el alcance del `MutexGuard` para no mantener bloqueos ni estados intermedios durante operaciones asíncronas, reduciendo riesgos de inconsistencias y bloqueos.

```
use std::sync::{Arc, Mutex};
use tokio::time::{sleep, Duration};

struct SharedState {
  processing: bool,
  value: i32,
}
#[tokio::main]
async fn main() {
  //-- Inicializa el estado compartido
  let state = Arc::new(Mutex::new(SharedState { processing: false, value: 0 }));

  //-- Clona la referencia para simular concurrencia
  let state_clone = Arc::clone(&state);

  let handle = tokio::spawn(async move {
    //-- Modifica el estado dentro de un bloque limitado
    {
      let mut data = state_clone.lock().unwrap();
      data.processing = true;
      data.value = 42;
      //-- Restaura la invariante antes de liberar el lock
      data.processing = false;
    }
    //-- El await ocurre sin mantener locks ni invariantes abiertas
    sleep(Duration::from_secs(2)).await;
  });

  //-- Espera a que termine la tarea
  handle.await.unwrap();

  //-- Verifica el estado final
```

```
    let final_state = state.lock().unwrap();
    println!("Processing: {}, Value: {}", final_state.processing, final_state.value);
}
```

2.6.7.1 Descripción

Los puntos `await` en código asíncrono representan ubicaciones donde la ejecución puede suspenderse y reanudarse posteriormente. Durante esta suspensión, el estado del programa puede cambiar de formas impredecibles, ya que otras tareas concurrentes pueden ejecutarse y modificar datos compartidos.

Esta regla establece que **no se deben mantener invariantes críticas** que dependan de condiciones externas a través de operaciones `await`. Una invariante crítica es una condición que debe permanecer verdadera para garantizar la correctitud del programa, como el estado de estructuras de datos, contadores de recursos, o relaciones entre variables.

El problema surge porque el tiempo transcurrido entre la suspensión y reanudación de una tarea asíncrona es indeterminado. Durante este período, otras partes del programa pueden ejecutarse y alterar el estado que se asumía constante antes del punto `await`.

Los riesgos asociados incluyen condiciones de carrera, corrupción de datos, inconsistencias en el estado del programa, y comportamientos impredecibles que pueden comprometer tanto la funcionalidad como la seguridad de la aplicación.

La mitigación efectiva requiere reevaluar las condiciones críticas después de cada operación `await`, utilizar primitivas de sincronización apropiadas para proteger el estado compartido, y diseñar algoritmos que no dependan de invariantes que puedan invalidarse durante la suspensión asíncrona.

Riesgos asociados

- Condiciones de carrera: otros hilos o tareas pueden modificar el estado compartido durante la suspensión, violando invariantes críticas.
- Corrupción de datos: los invariantes pueden romperse y dejar estructuras de datos en estados inconsistentes.
- Elevación de privilegios: actores maliciosos pueden manipular el estado mientras la invariante está incompleta, explotando la inconsistencia.
- Bloqueo indefinido o deadlocks: al restaurar un estado inconsistente tras el await, puede bloquearse la lógica del programa.
- Pérdida de integridad transaccional: operaciones que deberían ser atómicas pueden dividirse indebidamente, provocando fallos lógicos.
- Errores difíciles de depurar: los fallos por violación de invariantes a través de

await suelen ser intermitentes y complejos de identificar.

Buenas prácticas

- No mantener préstamos mutables activos a través de un `.await`; limitar el alcance antes del punto asíncrono.

```
//-- préstamo mutable mantenido a través de .await
async fn actualizar(data: &mut String) {
  let r = &mut *data;
  proceso_async().await;
  r.push_str("ok");
}
```

```
//-- limitar el alcance del préstamo antes de .await
async fn actualizar(data: &mut String) {
  {
    data.push_str("tmp");
  }
  proceso_async().await;
  data.push_str("ok");
}
```

- Evitar mantener bloqueos (`Mutex`, `RwLock`) durante un `.await`; liberar el candado antes de esperar.

```
//-- bloqueo mantenido mientras se espera en .await
async fn incrementar(shared: std::sync::Arc<tokio::sync::Mutex<i32>>) {
  let mut guard = shared.lock().await;
  proceso_async().await;
  *guard += 1;
}
```

```
//-- liberar el bloqueo antes del punto .await
async fn incrementar(shared: std::sync::Arc<tokio::sync::Mutex<i32>>) {
  {
    let mut guard = shared.lock().await;
    *guard += 1;
  }
  proceso_async().await;
}
```

- No depender de estados parciales en estructuras compartidas durante un `.await`; completar la transición antes de ceder el control.

```
//-- estado inconsistente expuesto durante .await
struct Sesion { activa: bool, token: Option<String> }
```

```
async fn iniciar(s: &mut Sesion) {
  s.activa = true;
  proceso_async().await;
  s.token = Some("abc".into());
}
```

```
//-- transición de estado completada antes de .await
async fn iniciar(s: &mut Sesion) {
  s.activa = true;
  s.token = Some("abc".into());
  proceso_async().await;
}
```

- Evitar referencias a datos internos que deban permanecer consistentes tras un `.await`; clonar o mover los datos necesarios.

```
//-- referencia a dato interno usada después de .await
async fn procesar(cfg: &mut Vec<String>) {
  let first = &cfg[0];
  proceso_async().await;
  println!("{}", first);
}
```

```
//-- clonar el dato antes del punto .await
async fn procesar(cfg: &mut Vec<String>) {
  let first = cfg[0].clone();
  proceso_async().await;
  println!("{}", first);
}
```

2.6.7.2 Estándares metodológicos

- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 3.3**: Lifetimes.
- **Rustonomicon 3.4**: Limits of Lifetimes.

2.6.8 [rust.sync.8] - Garantizar liberación de recursos aunque una tarea sea cancelada

ID	rust.sync.8	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La cancelación de tareas puede interrumpir la ejecución antes de que se complete la liberación de recursos, provocando **memory leaks** o el bloqueo indefinido de recursos del sistema. Los destructores automáticos no siempre garantizan la limpieza adecuada cuando se produce una cancelación asíncrona, especialmente con recursos como archivos, conexiones de red o mutex.

Codificación insegura

Gestión de un archivo abierto dentro de un hilo que puede finalizar antes de completar su trabajo. Se busca observar qué ocurre cuando el recurso no está protegido por un mecanismo automático de liberación y cómo una cancelación o salida anticipada puede provocar que el recurso no se cierre correctamente.

```
use std::fs::File;
use std::io::Write;
use std::mem;
use std::thread;

fn main() {
  //-- Se crea un archivo y se mueve al hilo
  let handle = thread::spawn(|| {
    //-- Apertura del archivo
    let mut file = File::create("no_liberado.txt").unwrap();

    //-- Escritura inicial
    writeln!(file, "Inicio de tarea").unwrap();

    //-- Se evita que Rust ejecute Drop automáticamente
    mem::forget(file);

    //-- Simulación de cancelación abrupta
    panic!("Tarea cancelada");
  });

  //-- Se espera el hilo (captura del panic)
  let _ = handle.join();

  //-- El archivo puede no haberse cerrado correctamente
  println!("Fin del programa");
```

```
}
```

Codificación recomendada

Uso del patrón RAII mediante una estructura con implementación de `Drop` para garantizar la liberación del recurso aunque la tarea termine por cancelación o error. Se demuestra cómo Rust ejecuta automáticamente la limpieza incluso cuando ocurre un `panic`.

```
use std::fs::File;
use std::io::Write;
use std::thread;

struct ArchivoSeguro {
  file: File,
}
impl ArchivoSeguro {
  //-- Constructor que abre el archivo
  fn new(path: &str) -> Self {
    let file = File::create(path).unwrap();
    ArchivoSeguro { file }
  }
  //-- Método para escribir datos
  fn escribir(&mut self, mensaje: &str) {
    writeln!(self.file, "{}", mensaje).unwrap();
  }
}
impl Drop for ArchivoSeguro {
  //-- Liberación automática del recurso
  fn drop(&mut self) {
    println!("Archivo cerrado correctamente");
  }
}
fn main() {
  //-- Se crea el hilo con gestión segura del recurso
  let handle = thread::spawn(|| {
    let mut archivo = ArchivoSeguro::new("liberado.txt");

    //-- Escritura inicial
    archivo.escribir("Inicio de tarea");

    //-- Simulación de cancelación abrupta
    panic!("Tarea cancelada");
  });

  //-- Se espera el hilo (captura del panic)
  let _ = handle.join();
```

```
    //-- Drop se ejecuta automáticamente pese al panic
    println!("Fin del programa");
}
```

2.6.8.1 Descripción

La cancelación de tareas asíncronas puede interrumpir la ejecución normal del código antes de que se complete la liberación de recursos. Esta situación genera riesgos de **resource leaks** cuando los recursos no se liberan correctamente tras la cancelación.

Los recursos que requieren liberación explícita incluyen descriptores de archivo, conexiones de red, mutex, memoria asignada dinámicamente y handles del sistema operativo. La cancelación abrupta de una tarea puede dejar estos recursos en un estado inconsistente o sin liberar.

El sistema de tipos proporciona mecanismos automáticos de limpieza a través del trait `Drop`, que se ejecuta cuando un valor sale de scope. Sin embargo, la cancelación de tareas puede interrumpir este proceso si no se implementa correctamente la gestión de recursos en contextos asíncronos.

La mitigación efectiva requiere el uso de **RAII patterns** combinados con manejo adecuado de señales de cancelación. Los tipos que implementan `Drop` garantizan la liberación automática de recursos cuando se destruyen correctamente.

Las primitivas de sincronización y los wrappers de recursos deben diseñarse para detectar cancelaciones y ejecutar rutinas de limpieza apropiadas. El uso de `select!` y otros mecanismos de control de flujo asíncrono debe considerar la liberación de recursos en todas las ramas de ejecución posibles.

La implementación de **graceful shutdown** permite que las tareas completen operaciones críticas de limpieza antes de terminar. Los timeouts y mecanismos de cancelación cooperativa proporcionan control sobre el proceso de terminación sin comprometer la integridad de los recursos.

Riesgos asociados

- Fugas de memoria por no liberar recursos heap asignados.
- Bloqueos (deadlocks) al dejar locks o mutex sin liberar.
- Fuga de descriptores de archivos o sockets abiertos que no se cierran adecuadamente.
- Corrupción del estado interno si no se restauran invariantes tras la cancelación de una tarea.
- Pérdida de datos importantes al no cerrar buffers o flujos correctamente.
- Dificultad en el diagnóstico de errores por estado inconsistente de recursos

compartidos.

- Vulnerabilidades de denegación de servicio (DoS) al agotar recursos del sistema (memoria, archivos, etc.).
- Exposición a ataques de escalada de privilegios si la propiedad de recursos sensibles no es liberada.

Buenas prácticas

- Implementar el trait `Drop` para garantizar la liberación automática de recursos al cancelarse una tarea.

```
//-- Recurso sin liberación explícita al cancelarse la tarea
struct Conexion {
  socket: TcpStream,
}
async fn procesar() {
  let _c = Conexion { socket: conectar() };
  operacion_larga().await;
}
```

```
//-- Liberación automática mediante implementación de Drop
struct Conexion {
  socket: TcpStream,
}
impl Drop for Conexion {
  fn drop(&mut self) {
    let _ = self.socket.shutdown(Shutdown::Both);
  }
}
async fn procesar() {
  let _c = Conexion { socket: conectar() };
  operacion_larga().await;
}
```

- Usar bloques `scope` o estructuras RAII para asegurar liberación determinística de recursos.

```
//-- Recurso creado fuera de un ámbito controlado
let archivo = File::create("datos.txt").unwrap();
tarea_cancelable().await;
procesar_archivo(archivo);
```

```
//-- Recurso limitado a un ámbito que garantiza liberación
{
  let archivo = File::create("datos.txt").unwrap();
  tarea_cancelable().await;
```

```
  procesar_archivo(&archivo);
}
```

- Gestionar cancelaciones en tareas asíncronas usando `tokio::select!` o mecanismos equivalentes para cerrar recursos explícitamente.

```
//-- Tarea sin manejo de cancelación explícita
async fn escuchar(mut stream: TcpStream) {
  stream.readable().await.unwrap();
  procesar(&mut stream).await;
}
```

```
//-- Manejo explícito de cancelación y cierre seguro
async fn escuchar(mut stream: TcpStream, cancel: CancellationToken) {
  tokio::select! {
    _ = stream.readable() => {
      procesar(&mut stream).await;
    }
    _ = cancel.cancelled() => {
      let _ = stream.shutdown(Shutdown::Both);
    }
  }
}
```

- Evitar `mem::forget` o fugas intencionales que impidan la liberación automática en escenarios de cancelación.

```
//-- Uso de mem::forget que impide liberar el recurso
let conexion = Conexion::nueva();
std::mem::forget(conexion);
```

```
//-- Liberación automática permitiendo que Drop se ejecute
let conexion = Conexion::nueva();
drop(conexion);
```

2.6.8.2 Estándares metodológicos

- **Rust-8.DTOR-FAIL**: Destructors never fail.
- **Rust-8.DTOR-BLOCK**: Destructors that may block have alternatives.
- **Rustonomicon 6.2**: Destructors.

2.7 Uso de unsafe y FFI

2.7.1 [rust.uns.1] - Encapsular todo código unsafe dentro de abstracciones seguras con API pública libre de unsafe

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

El código `unsafe` permite operaciones de bajo nivel que evaden las garantías de seguridad del compilador, como acceso directo a memoria o llamadas a funciones externas. Cuando se utiliza `unsafe`, es necesario **encapsularlo dentro de abstracciones seguras** que expongan únicamente interfaces públicas sin bloques `unsafe`, evitando que el código cliente tenga acceso directo a operaciones potencialmente peligrosas.

Codificación insegura

Uso directo de código `unsafe` expuesto en la API pública. Se muestra cómo una variable global mutable requiere bloques `unsafe` y cómo, al no encapsularla, obliga a que cualquier consumidor también utilice `unsafe`, aumentando el riesgo de uso incorrecto.

```
static mut COUNTER: i32 = 0;

pub unsafe fn increment() {
  //-- Incrementa el contador global sin ninguna encapsulación
  COUNTER += 1;
}
pub unsafe fn get() -> i32 {
  //-- Devuelve el valor actual del contador global
  COUNTER
}
fn main() {
  //-- El llamador está obligado a usar unsafe porque la API lo expone
  unsafe {
    increment();
    println!("Valor del contador: {}", get());
  }
}
```

Codificación recomendada

Encapsulación del código `unsafe` dentro de una abstracción segura. El acceso a la variable global mutable se limita a la implementación interna, mientras que la API pública no expone `unsafe`, reduciendo el riesgo de uso indebido por parte del consumidor.

```
static mut COUNTER: i32 = 0;

pub struct Counter;

impl Counter {
  //-- Método público seguro que encapsula el bloque unsafe
  pub fn increment() {
    unsafe {
      COUNTER += 1;
    }
  }
  //-- Método público seguro para obtener el valor actual
  pub fn get() -> i32 {
    unsafe { COUNTER }
  }
}
fn main() {
  //-- El llamador usa una API completamente segura
  Counter::increment();
  println!("Valor del contador: {}", Counter::get());
}
```

2.7.1.1 Descripción

La regla establece que todo código `unsafe` debe quedar contenido dentro de **abstracciones seguras** que expongan únicamente interfaces públicas sin bloques `unsafe`. Esto significa que los módulos o tipos que internamente utilicen operaciones no seguras deben proporcionar una API pública que mantenga todas las invariantes de seguridad sin requerir que el código cliente maneje directamente las complejidades del código `unsafe`.

Los riesgos asociados al incumplimiento de esta regla incluyen la propagación de vulnerabilidades de seguridad de memoria a través de la base de código, la dificultad para localizar y corregir errores relacionados con operaciones no seguras, y la pérdida de las garantías de seguridad que caracterizan al lenguaje.

La mitigación efectiva requiere diseñar cuidadosamente las interfaces públicas para que mantengan todas las precondiciones necesarias, implementar validaciones exhaustivas en los límites de las abstracciones, y documentar claramente las invariantes que debe cumplir el código `unsafe` interno. Las abstracciones deben garantizar que ninguna combinación válida de llamadas a su API pública pueda resultar en comportamiento indefinido o violaciones de seguridad de memoria.

Riesgos asociados

- Ejecución de código inseguro desde partes inadvertidas del programa, exponiendo a vulnerabilidades de memoria.

- Violaciones de garantías de seguridad de Rust, como aliasing ilegal o desreferenciación de punteros nulos.
- Introducción de condiciones de carrera y errores de concurrencia que pueden causar comportamientos indefinidos.
- Dificultad para auditar la base de código, incrementando la posibilidad de que fallos de seguridad pasen desapercibidas.
- Escape accidental de punteros o referencias no inicializadas, llevando a corrupción de memoria y fallos en tiempo de ejecución.
- Incremento del riesgo de escalada de privilegios o ataques de ejecución arbitraria de código.
- Posibilidad de causar pánico o crash del programa en producción debido a acceso inseguro a recursos.
- La API pública contaminada con `unsafe` fuerza a los usuarios a utilizar `unsafe`, propagando malas prácticas.

Buenas prácticas

- Encapsular bloques `unsafe` dentro de funciones privadas y exponer únicamente funciones públicas seguras.

```
//-- Uso directo de punteros en API pública
pub unsafe fn read_value(ptr: *const i32) -> i32 {
  *ptr
}
```

```
//-- Encapsulamiento interno y validación previa
pub fn read_value(slice: &[i32], index: usize) -> Option<i32> {
  if index < slice.len() {
    Some(unsafe { *slice.get_unchecked(index) })
  } else {
    None
  }
}
```

- Crear tipos abstracción (newtype) que garanticen invariantes antes de ejecutar código `unsafe`.

```
//-- Acceso inseguro sin garantizar invariantes
pub unsafe fn create_slice(ptr: *const u8, len: usize) -> &'static [u8] {
  std::slice::from_raw_parts(ptr, len)
}
```

```
//-- Tipo que valida condiciones antes de exponer datos
pub struct SafeSlice<'a> {
  data: &'a [u8],
}
impl<'a> SafeSlice<'a> {
  pub fn new(data: &'a [u8]) -> Self {
    SafeSlice { data }
  }
  pub fn get(&self, index: usize) -> Option<u8> {
    if index < self.data.len() {
      Some(unsafe { *self.data.get_unchecked(index) })
    } else {
      None
    }
  }
}
```

- Limitar el alcance del bloque `unsafe` al mínimo necesario dentro de la función.

```
//-- Bloque unsafe abarca más código del necesario
pub fn increment(ptr: *mut i32) {
  unsafe {
    *ptr += 1;
    println!("{}", *ptr);
  }
}
```

```
//-- Bloque unsafe reducido a la operación crítica
pub fn increment(ptr: *mut i32) {
  unsafe { *ptr += 1; }
  println!("{}", unsafe { *ptr });
}
```

- Documentar y reforzar explícitamente las precondiciones que justifican el uso interno de `unsafe`.

```
//-- Uso inseguro sin validación previa
pub fn write_value(ptr: *mut i32, value: i32) {
  unsafe { *ptr = value; }
}
```

```
//-- Validación explícita antes del acceso inseguro
pub fn write_value(ptr: *mut i32, value: i32) -> Result<(), &'static str> {
  if ptr.is_null() {
    return Err("null pointer");
  }
  unsafe { *ptr = value; }
```

```
  Ok(())
}
```

2.7.1.2 Estándares metodológicos

- **Rust-6.OBJECT**: Traits are object-safe if they may be useful as a trait object.
- **Rust-7.NEWTYPE**: Newtypes provide static distinctions.
- **Rustonomicon 1.1**: How Safe and Unsafe Interact.
- **Rustonomicon 1.3**: Working with Unsafe.

2.7.2 [rust.uns.2] - Minimizar el tamaño, alcance y frecuencia de cada bloque unsafe

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

Los bloques `unsafe` permiten operaciones que el compilador no puede verificar automáticamente, introduciendo riesgos de **corrupción de memoria** y **comportamiento indefinido**. El uso extensivo o innecesario de estos bloques compromete las garantías de seguridad del lenguaje y aumenta la superficie de ataque para vulnerabilidades críticas.

Codificación insegura

Manipulación de memoria cruda mediante punteros para convertir un array en un slice. Se busca mostrar cómo el alcance del bloque `unsafe` influye en la exposición a errores y cómo su tamaño puede afectar la seguridad general del programa.

```
use std::slice;

fn main() {
  //-- Se declara un array estático
  let data = [1, 2, 3, 4];

  //-- Se obtiene un puntero crudo al array
  let ptr = data.as_ptr();
  let len = data.len();

  //-- Se abre un bloque unsafe más grande de lo necesario
  unsafe {
    //-- Conversión del puntero crudo a slice
    let slice = slice::from_raw_parts(ptr, len);
```

```
    //-- Operaciones adicionales que no requieren unsafe
    for value in slice {
      println!("Valor: {}", value);
    }
    //-- Lógica adicional innecesariamente dentro de unsafe
    let sum: i32 = slice.iter().sum();
    println!("Suma: {}", sum);
  }
}
```

Codificación recomendada

Manipulación de memoria cruda mediante punteros para convertir un array en un slice, limitando el bloque `unsafe` exclusivamente a la operación que realmente lo requiere. Se pretende demostrar cómo reducir tamaño y alcance mejora la claridad y disminuye riesgos.

```
use std::slice;

fn main() {
  //-- Se declara un array estático
  let data = [1, 2, 3, 4];

  //-- Se obtiene un puntero crudo al array
  let ptr = data.as_ptr();
  let len = data.len();

  //-- Se limita unsafe únicamente a la conversión necesaria
  let slice = unsafe { slice::from_raw_parts(ptr, len) };

  //-- El resto del código permanece en contexto seguro
  for value in slice {
    println!("Valor: {}", value);
  }
  //-- Operaciones seguras fuera de unsafe
  let sum: i32 = slice.iter().sum();
  println!("Suma: {}", sum);
}
```

2.7.2.1 Descripción

Los bloques `unsafe` permiten realizar operaciones que el compilador no puede verificar automáticamente, como la desreferenciación de punteros sin procesar, el acceso a variables estáticas mutables o la llamada a funciones externas. Estas operaciones introducen riesgos de seguridad de memoria que normalmente están ausentes en el código seguro.

La regla establece que debemos **minimizar el tamaño, alcance y frecuencia** de cada bloque `unsafe`. Esta práctica reduce la superficie de ataque donde pueden ocurrir errores de memoria y facilita la revisión y auditoría del código crítico.

El **tamaño** se refiere al número de líneas de código dentro del bloque. Mantener bloques pequeños permite identificar más fácilmente la fuente de problemas potenciales y simplifica el análisis de corrección.

El **alcance** hace referencia a la cantidad de variables y recursos que el bloque `unsafe` puede afectar. Limitar el alcance reduce las dependencias e interacciones complejas que pueden generar comportamientos impredecibles.

La **frecuencia** indica cuántas veces aparecen estos bloques en el código base. Reducir su número disminuye las oportunidades de introducir vulnerabilidades y errores de memoria.

Los riesgos asociados incluyen corrupción de memoria, accesos fuera de límites, condiciones de carrera y comportamiento indefinido. Estos problemas pueden comprometer la estabilidad y seguridad de la aplicación.

Las estrategias de mitigación incluyen encapsular la funcionalidad `unsafe` en interfaces seguras, documentar exhaustivamente las invariantes que deben mantenerse, y aplicar revisiones de código rigurosas. También resulta efectivo extraer la lógica `unsafe` a funciones auxiliares pequeñas y bien definidas que puedan ser probadas de forma aislada.

Riesgos asociados

- Introducción de vulnerabilidades de seguridad debido a violación de garantías de seguridad y memoria (ej. desbordamientos de buffer, uso después de liberar, referencias colgantes).
- Dificultad para auditar y revisar el código, aumentando la probabilidad de errores o comportamientos inesperados en bloques grandes o mal delimitados.
- Exposición innecesaria de áreas críticas a operaciones inseguras, elevando la superficie de ataque.
- Incremento en la complejidad del código, dificultando su comprensión y mantenimiento.
- Propagación inadvertida de invariantes rotas o errores a otras partes del programa fuera del bloque `unsafe`.
- Reducción de la efectividad de herramientas automáticas (ej. linters, analizadores estáticos) para detectar problemas dentro de bloques extensos o frecuentes.
- Mayor probabilidad de incumplir otros principios de seguridad, como la separación de privilegios y el principio de menor exposición.

Buenas prácticas

- Encapsular el código `unsafe` dentro de funciones seguras con validaciones previas.

```
//-- Uso directo de unsafe sin validaciones previas
let ptr = get_ptr();
unsafe {
  *ptr = 10;
}
```

```
//-- Encapsulación en función segura con validación antes del bloque unsafe
fn set_value(ptr: *mut i32) {
  if ptr.is_null() { return; }
  unsafe {
    *ptr = 10;
  }
}
```

- Reducir el alcance del bloque `unsafe` únicamente a la operación estrictamente necesaria.

```
//-- Bloque unsafe abarcando más código del necesario
unsafe {
  let value = *ptr;
  println!("{}", value);
}
```

```
//-- Bloque unsafe limitado solo a la desreferenciación
let value = unsafe { *ptr };
println!("{}", value);
```

- Evitar repetir bloques `unsafe` frecuentes centralizando la lógica en una única abstracción controlada.

```
//-- Múltiples bloques unsafe repetidos en diferentes puntos
let a = unsafe { *ptr };
let b = unsafe { *ptr.add(1) };
```

```
//-- Función auxiliar que concentra el acceso inseguro
fn read_offset(ptr: *const i32, offset: usize) -> i32 {
  unsafe { *ptr.add(offset) }
}
let a = read_offset(ptr, 0);
let b = read_offset(ptr, 1);
```

- Documentar claramente las invariantes que justifican el uso de `unsafe` para limitar su uso indebido.

```
//-- Uso de unsafe sin especificar las condiciones requeridas
unsafe {
  buffer.set_len(10);
}
```

```
//-- Validación explícita antes de ajustar la longitud manualmente
if buffer.capacity() >= 10 {
  unsafe {
    buffer.set_len(10);
  }
}
```

2.7.2.2 Estándares metodológicos

- **Rust-6.INTERMEDIATE**: Functions expose intermediate results to avoid duplicate work.
- **Rustonomicon 1.2**: What Unsafe Can Do.
- **Rustonomicon 1.3**: Working with Unsafe.

2.7.3 [rust.uns.3] - Validar exhaustivamente toda entrada proveniente de FFI antes de su uso

ID	**rust.uns.3**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las interfaces de función externa (FFI) permiten interactuar con código escrito en otros lenguajes, pero esta interoperabilidad introduce **riesgos de seguridad significativos**. Los datos que atraviesan estas fronteras pueden contener valores maliciosos, corruptos o simplemente incompatibles con las expectativas del programa receptor. Sin validación adecuada, estos datos pueden provocar **comportamientos indefinidos**, vulnerabilidades de memoria o fallos del sistema.

Codificación insegura

Se muestra una función expuesta vía FFI que recibe un puntero C y lo convierte directamente en cadena Rust para procesarlo. La intención es ilustrar cómo se suele integrar código Rust con bibliotecas externas en C y qué riesgos aparecen cuando no se validan correctamente punteros, codificación y tamaño de datos antes de usarlos.

```
use std::ffi::{CStr, CString};
```

```
use std::os::raw::c_char;

unsafe extern "C" fn procesar_entrada(ptr: *const c_char) {
  //-- Se asume que el puntero es válido y no es nulo
  let c_str = CStr::from_ptr(ptr);

  //-- Se asume que el contenido es UTF-8 válido
  let mensaje = c_str.to_str().unwrap();

  //-- Se usa directamente la entrada sin validar tamaño ni contenido
  println!("Mensaje recibido: {}", mensaje);
}
fn main() {
  //-- Simula una llamada desde código C
  let entrada = CString::new("Hola desde C").unwrap();

  unsafe {
    procesar_entrada(entrada.as_ptr());
  }
}
```

Codificación recomendada

Se presenta una función expuesta vía FFI que recibe un puntero C, pero aplicando validaciones estrictas antes de utilizar los datos: verificación de puntero nulo, comprobación de UTF-8 válido y restricción de tamaño. El objetivo es aprender a reducir riesgos de corrupción de memoria, denegación de servicio y fallos por datos malformados provenientes de código externo.

```
use std::ffi::{CStr, CString};
use std::os::raw::c_char;

const MAX_LONGITUD: usize = 64;

unsafe extern "C" fn procesar_entrada(ptr: *const c_char) {
  //-- Validar puntero nulo antes de usarlo
  if ptr.is_null() {
    eprintln!("Error: puntero nulo recibido");
    return;
  }
  //-- Convertir a CStr de forma controlada
  let c_str = CStr::from_ptr(ptr);

  //-- Validar que sea UTF-8 válido
  let mensaje = match c_str.to_str() {
    Ok(valor) => valor,
    Err(_) => {
      eprintln!("Error: entrada no es UTF-8 válida");
```

```
      return;
    }
  };

  //-- Validar longitud máxima permitida
  if mensaje.len() > MAX_LONGITUD {
    eprintln!("Error: entrada demasiado larga");
    return;
  }
  //-- Uso seguro tras validaciones
  println!("Mensaje validado: {}", mensaje);
}
fn main() {
  //-- Simula una llamada desde código C
  let entrada = CString::new("Hola desde C").unwrap();

  unsafe {
    procesar_entrada(entrada.as_ptr());
  }
}
```

2.7.3.1 Descripción

La interfaz de funciones externas (FFI) permite la interoperabilidad con código escrito en otros lenguajes, principalmente C y C++. Esta capacidad introduce riesgos de seguridad significativos, ya que el código externo no está sujeto a las garantías de seguridad de memoria que proporciona el compilador.

Los datos que atraviesan la frontera FFI pueden contener valores maliciosos o malformados que comprometan la integridad del programa. El código externo puede generar punteros inválidos, estructuras de datos corruptas, cadenas sin terminación nula, o valores fuera de los rangos esperados. Estos datos no validados pueden provocar comportamiento indefinido, corrupción de memoria, o vulnerabilidades explotables.

La validación exhaustiva debe aplicarse a todos los tipos de datos recibidos: punteros, estructuras, arrays, cadenas de caracteres, y valores primitivos. Los punteros requieren verificación de nulidad y validez antes de su desreferenciación. Las estructuras deben validar la coherencia de sus campos y el cumplimiento de invariantes. Los arrays necesitan comprobación de límites y longitudes. Las cadenas requieren verificación de codificación y terminación adecuada.

La mitigación efectiva incluye la implementación de funciones de validación específicas para cada tipo de dato recibido desde FFI. Estas funciones deben verificar rangos, formatos, y restricciones semánticas antes de permitir el uso de los datos. La conversión a tipos seguros nativos debe realizarse únicamente después de completar la validación. El uso de wrappers seguros alrededor de las funciones FFI facilita la centralización de las validaciones y reduce la superficie de ataque.

La documentación clara de las precondiciones y postcondiciones de las funciones FFI ayuda a mantener la consistencia en las validaciones. El establecimiento de políticas de manejo de errores específicas para datos inválidos permite una respuesta controlada ante intentos de explotación.

Riesgos asociados

- Ejecución de código arbitrario debido a datos maliciosos provenientes de bibliotecas externas o código no Rust.
- Desbordamientos de buffer por recibir datos con longitudes no controladas ni validadas.
- Vulnerabilidades de corrupción de memoria, incluido uso de punteros nulos o inválidos.
- Condiciones de carrera si los datos compartidos desde FFI no son validados correctamente.
- Exposición a ataques de inyección de código, comandos o datos estructurados como SQL o código de script.
- Lectura o escritura fuera de límites, por confiar en arrays, cadenas o buffers sin validación de su tamaño real.
- Fugas de información sensible por procesar datos corruptos o inesperados.
- Pánico en tiempo de ejecución y posible denegación de servicio (DoS) por recibir datos inadecuados o inesperados.
- Desbordamientos de enteros o conversiones inválidas que pueden realizar operaciones inesperadas o inseguras.
- Violaciones de invariantes de seguridad, comprometiendo la seguridad global de toda la aplicación.

Buenas prácticas

- Verificar punteros recibidos por FFI antes de desreferenciarlos y manejar adecuadamente valores nulos.

```
//-- Desreferencia directa de puntero externo sin validación
pub unsafe extern "C" fn procesar(ptr: *const i32) -> i32 {
  *ptr
}
```

```
//-- Validación explícita de puntero nulo antes de usarlo
pub unsafe extern "C" fn procesar(ptr: *const i32) -> i32 {
  if ptr.is_null() {
    return -1;
```

```
  }
  *ptr
}
```

- Validar la longitud y límites de buffers recibidos desde FFI antes de crear slices.

```
//-- Creación de slice desde puntero externo sin validar longitud
pub unsafe extern "C" fn leer_buffer(ptr: *const u8, len: usize) {
  let datos = std::slice::from_raw_parts(ptr, len);
  println!("{:?}", datos);
}
```

```
//-- Verificación de puntero y límites antes de crear el slice
pub unsafe extern "C" fn leer_buffer(ptr: *const u8, len: usize) {
  if ptr.is_null() || len > 1024 {
    return;
  }
  let datos = std::slice::from_raw_parts(ptr, len);
  println!("{:?}", datos);
}
```

- Comprobar que cadenas recibidas desde FFI estén correctamente terminadas y sean UTF-8 válidas antes de convertirlas.

```
//-- Conversión directa de C string sin validar contenido
pub unsafe extern "C" fn mostrar(c_str: *const i8) {
  let s = std::ffi::CStr::from_ptr(c_str).to_str().unwrap();
  println!("{}", s);
}
```

```
//-- Validación de puntero y manejo seguro de errores de conversión
pub unsafe extern "C" fn mostrar(c_str: *const i8) {
  if c_str.is_null() {
    return;
  }
  if let Ok(s) = std::ffi::CStr::from_ptr(c_str).to_str() {
    println!("{}", s);
  }
}
```

- Validar valores numéricos o códigos recibidos desde FFI antes de usarlos en lógica interna o conversiones.

```
//-- Uso directo de valor externo como índice sin validación
pub unsafe extern "C" fn acceder(idx: usize) -> i32 {
  let datos = [10, 20, 30];
```

```
  datos[idx]
}
```

```
//-- Validación del rango antes de acceder al array
pub unsafe extern "C" fn acceder(idx: usize) -> i32 {
  let datos = [10, 20, 30];
  if idx >= datos.len() {
    return -1;
  }
  datos[idx]
}
```

2.7.3.2 Estándares metodológicos

- **Rust-2.CONV-TRAITS**: Conversions use the standard traits From, AsRef, AsMut.
- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 4.3**: Casts.
- **Rustonomicon 4.4**: Transmutes.

2.7.4 [rust.uns.4] - No asumir inicialización, alineación ni validez de memoria recibida desde código externo

ID	rust.uns.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La interoperabilidad con código externo introduce riesgos significativos cuando se asumen propiedades sobre la memoria que este proporciona. El código externo puede entregar **memoria no inicializada**, con **alineación incorrecta** o que apunte a **direcciones inválidas**, lo que puede provocar comportamiento indefinido, corrupción de datos o fallos de segmentación.

Codificación insegura

Procesamiento de un buffer recibido desde código externo mediante FFI. Se muestra cómo se accede directamente a un puntero crudo recibido como parámetro y se convierte en una referencia sin realizar validaciones previas. La intención es entender los riesgos de asumir que la memoria apuntada está inicializada, alineada y es válida.

```
use std::os::raw::c_uint;

//-- Función expuesta para ser llamada desde código externo
#[no_mangle]
```

```
pub extern "C" fn procesar_datos(ptr: *const c_uint, len: usize) -> u32 {
  //-- Se asume que el puntero es válido y está correctamente alineado
  //-- Se crea un slice directamente desde el puntero sin validaciones
  let datos = unsafe { std::slice::from_raw_parts(ptr, len) };

  //-- Se accede al primer elemento asumiendo que existe y está inicializado
  datos[0]
}
fn main() {
  //-- Simulación de llamada externa con puntero nulo
  //-- Esto compila, pero en ejecución provoca comportamiento indefinido
  let resultado = procesar_datos(std::ptr::null(), 1);

  println!("Resultado: {}", resultado);
}
```

Codificación recomendada

Procesamiento seguro de un buffer recibido desde código externo validando condiciones mínimas antes de usar memoria no confiable. Se comprueba que el puntero no sea nulo, que esté alineado correctamente y que la longitud sea coherente antes de crear el slice y acceder a los datos.

```
use std::mem;
use std::os::raw::c_uint;

//-- Función expuesta para ser llamada desde código externo
#[no_mangle]
pub extern "C" fn procesar_datos(ptr: *const c_uint, len: usize) -> Option<u32> {
  //-- Validar que el puntero no sea nulo
  if ptr.is_null() {
    return None;
  }
  //-- Validar alineación según el tipo esperado
  if (ptr as usize) % mem::align_of::<c_uint>() != 0 {
    return None;
  }
  //-- Validar que la longitud sea mayor que cero
  if len == 0 {
    return None;
  }
  //-- Crear el slice solo después de validar condiciones básicas
  let datos = unsafe { std::slice::from_raw_parts(ptr, len) };

  //-- Acceder al primer elemento de forma controlada
  datos.get(0).copied()
}
fn main() {
```

```
    //-- Crear datos válidos en memoria Rust
    let valores: Vec<c_uint> = vec![10, 20, 30];

    //-- Pasar puntero y longitud válidos
    let resultado = procesar_datos(valores.as_ptr(), valores.len());

    match resultado {
      Some(valor) => println!("Resultado: {}", valor),
      None => println!("Entrada inválida"),
    }
}
```

2.7.4.1 Descripción

La memoria recibida desde código externo no garantiza inicialización correcta, alineación adecuada ni validez de los datos contenidos. Esta situación se presenta frecuentemente en la interoperabilidad con bibliotecas escritas en C, llamadas al sistema operativo, comunicación entre procesos y acceso a dispositivos de hardware.

El problema surge porque el sistema de tipos no puede verificar automáticamente las propiedades de memoria que proviene de fuentes externas. Los datos pueden contener valores no inicializados, estar ubicados en direcciones no alineadas para el tipo de dato esperado, o apuntar a regiones de memoria inválidas o liberadas.

Los riesgos asociados incluyen comportamiento indefinido, corrupción de memoria, fallos de segmentación y vulnerabilidades de seguridad. La lectura de memoria no inicializada puede exponer información sensible, mientras que el acceso a memoria mal alineada puede causar excepciones en ciertas arquitecturas.

La mitigación requiere validación explícita de todos los aspectos de la memoria externa antes de su uso. Esto incluye verificar la alineación mediante funciones específicas, validar que los punteros apunten a regiones de memoria accesibles e inicializar explícitamente los datos cuando sea necesario. Las funciones `unsafe` deben encapsular estas validaciones y proporcionar interfaces seguras para el resto del código.

La documentación debe especificar claramente los contratos de memoria esperados, incluyendo requisitos de alineación, estado de inicialización y tiempo de vida válido de los datos externos.

Riesgos asociados

- Lectura de memoria no inicializada, conduciendo a resultados indefinidos o fuga de información sensible.
- Desbordamientos de buffer derivados de una alineación incorrecta, lo que permite vulnerabilidades como corrupción de memoria.
- Violaciones de segmentación debido al acceso a direcciones de memoria

inválidas o no asignadas.

- Ejecución de código arbitrario si memoria mal alineada o inválida es utilizada en operaciones inseguras.
- Fallos de programa (crash) inesperados por manipulación incorrecta de punteros recibidos externamente.
- Confusión de tipos que puede desembocar en corrupción de datos o ejecución de comportamientos no definidos por el estándar.
- Dificultad en depuración y mantenimiento, dado que los errores pueden manifestarse de manera errática y ser difíciles de rastrear.

Buenas prácticas

- Validar punteros recibidos desde código externo antes de dereferenciarlos y comprobar que no sean nulos.

```
//-- Uso directo de puntero externo sin validación
  unsafe fn procesar(ptr: *const i32) {
    let valor = *ptr;
    println!("{}", valor);
  }
```

```
//-- Validación explícita de puntero antes de dereferenciar
  unsafe fn procesar(ptr: *const i32) {
    if ptr.is_null() {
      return;
    }
    let valor = *ptr;
    println!("{}", valor);
  }
```

- No asumir alineación correcta de memoria externa; utilizar funciones que permitan lectura sin requisitos de alineación.

```
//-- Lectura directa asumiendo alineación correcta
  unsafe fn leer(ptr: *const u32) -> u32 {
    *ptr
  }
```

```
//-- Lectura utilizando read_unaligned para evitar fallos por alineación
  use std::ptr;
  unsafe fn leer(ptr: *const u32) -> u32 {
    ptr::read_unaligned(ptr)
  }
```

- No asumir que el tamaño o longitud de memoria externa es válido; validar límites antes de crear slices.

```
//-- Creación de slice confiando en longitud externa
  unsafe fn crear_slice(ptr: *const u8, len: usize) -> &'static [u8] {
    std::slice::from_raw_parts(ptr, len)
  }
```

```
//-- Validación básica de puntero y longitud antes de crear slice
  unsafe fn crear_slice(ptr: *const u8, len: usize, max: usize) -> Option<&'static
[u8]> {
    if ptr.is_null() || len > max {
      return None;
    }
    Some(std::slice::from_raw_parts(ptr, len))
  }
```

- No asumir que los datos externos contienen valores válidos para tipos de alto nivel; validar antes de convertir.

```
//-- Conversión directa de bytes externos a &str sin validación
  unsafe fn convertir(ptr: *const u8, len: usize) -> &'static str {
    let slice = std::slice::from_raw_parts(ptr, len);
    std::str::from_utf8_unchecked(slice)
  }
```

```
//-- Validación de UTF-8 antes de convertir a &str
  unsafe fn convertir(ptr: *const u8, len: usize) -> Option<&'static str> {
    if ptr.is_null() {
      return None;
    }
    let slice = std::slice::from_raw_parts(ptr, len);
    std::str::from_utf8(slice).ok()
  }
```

2.7.4.2 Estándares metodológicos

- **Rust-8.VALIDATE**: Functions validate their arguments.
- **Rustonomicon 2.1**: repr(Rust).
- **Rustonomicon 5.1**: Checked.

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