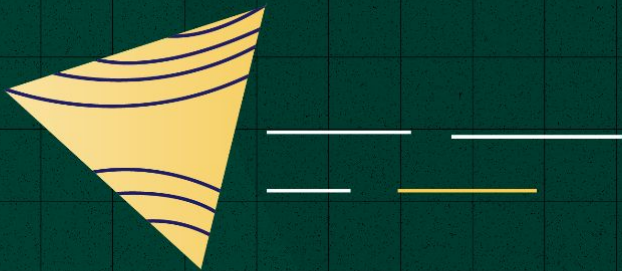




# Curso de Introducción a Seguridad de Smart Contracts







**Bienvenida**



# Sebastián Pérez



@sebaleoperez  
@blockacademycl



# Importancia de la seguridad en el desarrollo de contratos

# ■ ¿Por qué es importante la seguridad?

- Inmutabilidad
- Incentivos económicos
- Confianza







# Buenas prácticas





# Correcto uso del gas





**Controlar el gas  
es controlar el uso  
del contrato**

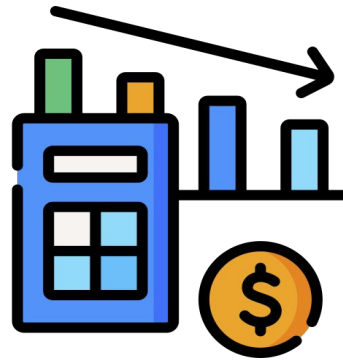


# Controlar el gas

**Una buena gestión del gas nos ayuda también a:**



Mejorar la  
performance



Reducir costos



Ejecutar código con  
malas intenciones





# Utilización de librerías



# **No reinventar la rueda**

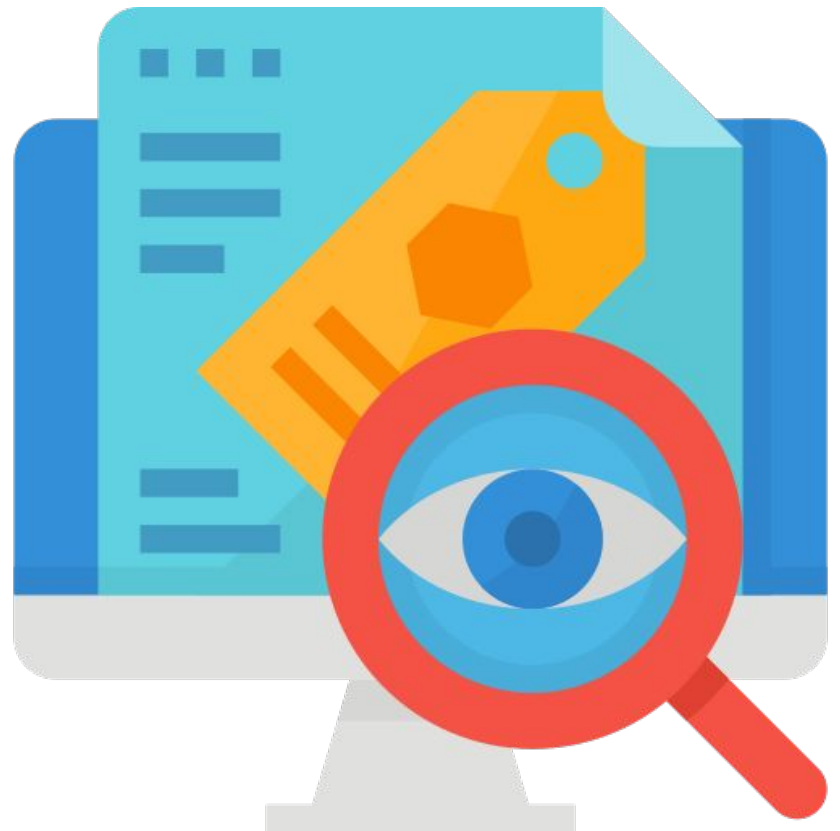
Las librerías nos ofrecen soluciones a desafíos que podemos encontrar en nuestros proyectos.





# Soluciones avaladas

Los códigos correspondientes a librerías ya han sido probados y auditados, lo cual hace que estos contratos sean confiables para su utilización.







# Control de acceso



# ■ Prevenir es mejor que curar

Controlar quién accede a las funciones de un contrato puede evitar que un usuario con malas intenciones vulnere o dañe nuestro contrato.





# ■ Bloquear un contrato

El control de acceso también puede bloquear un contrato en caso de detectar un ataque y así poder tomar medidas para salvarlo.







**Ejercer el control de un contrato puede convertir nuestro proyecto en un escenario centralizado.**

**Control = Centralización**



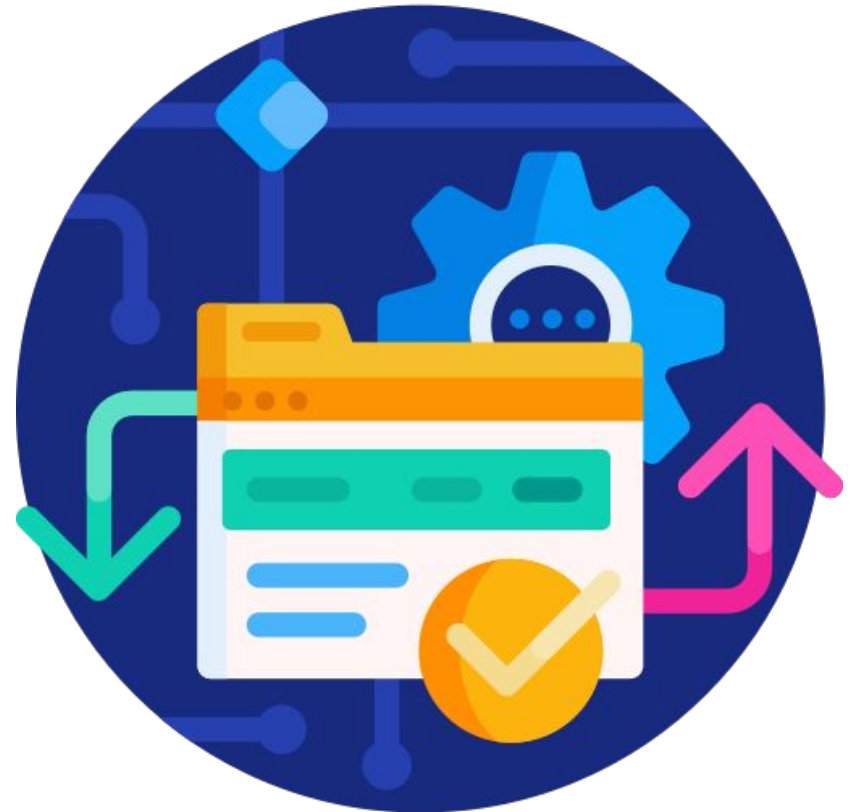


# Transferencias y Hooks



# La importancia de los fundamentos

Conocer a fondo el funcionamiento de cómo enviar Ether desde un contrato y cómo un contrato recibe transferencias es clave a la hora de detectar posibles vulnerabilidades.







# **Vulnerabilidades con variables**





# Problema con tx.origin



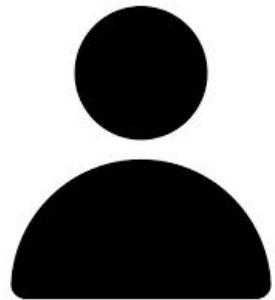
## Diferencias con msg.sender

- tx.origin = origen de la transacción
- msg.sender = emisor del mensaje



# Imaginemos esta situación

El usuario A crea un contrato al cual solo puede acceder con su cuenta por medio del control de acceso.





# ■ Imaginemos esta situación

El usuario *B* crea un contrato malicioso y engaña al usuario *A* para que lo llame.





# ■ Imaginemos esta situación

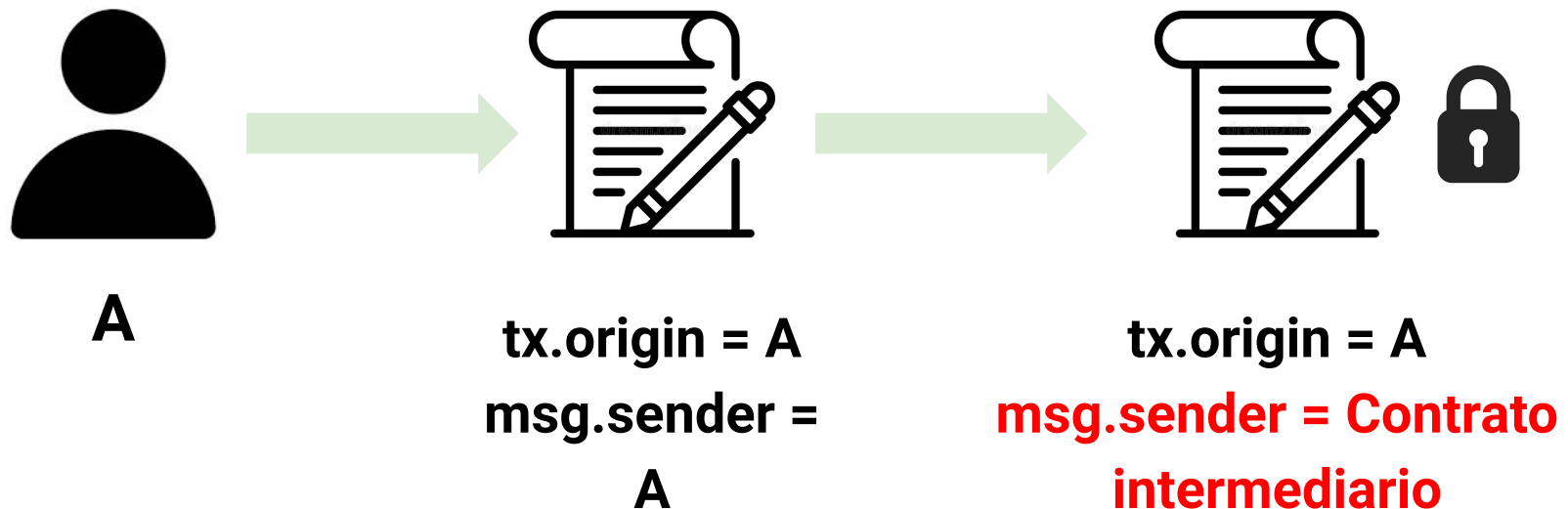
El contrato intermediario llama al contrato protegido. ¿Qué valores tienen las variables?





# Imaginemos esta situación

El contrato intermediario llama al contrato protegido. ¿Qué valores tienen las variables?







# Dependencia con timestamp





# Flujos críticos con timestamp

Muchas veces se utiliza el timestamp del bloque actual como entrada de una operación crítica como determinar el ganador de un sorteo.



# Debemos evitar datos manipulables/predecibles

- `block.timestamp`
- `block.number`







# **Vulnerabilidades del almacenamiento**





# Overflow y underflow



# El problema de los límites



# **Compatibilidad hacia atrás**

Este problema fue resuelto en las últimas versiones de Solidity. Sin embargo, si trabajamos con un compilador más antiguo debemos tener en cuenta este problema.





# Variables privadas

# Nada es realmente privado

Toda variable que se almacene en el contrato es visible independientemente de su modificador.

Solo basta, saber su posición en el almacenamiento para accederla.





# **Problemas con llamadas externas**



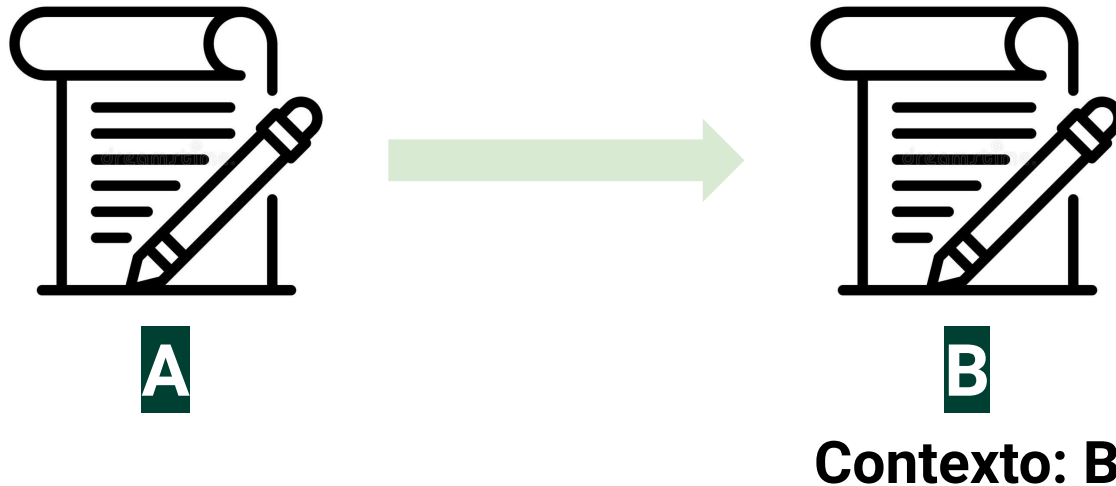


# DelegateCall



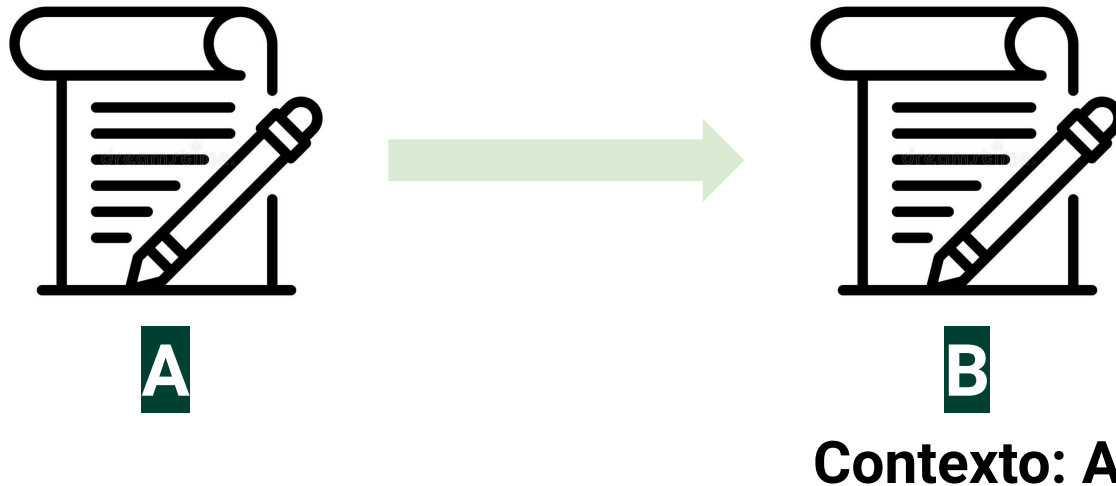
## Contexto de Call

Las llamadas externas con *Call* se ejecutan en el contexto del contrato receptor.



# Contexto de DelegateCall

Las llamadas externas con *DelegateCall* se ejecutan en el contexto del contrato llamador.







# Gas insuficiente

## Gas insuficiente

Puede que en nuestro código tengamos asignación de variables y luego una llamada externa (call). Si no controlamos la salida de la llamada puede darse el caso de que la función externa no se ejecute por falta de gas pero sí el resto del código.





# **Ataques con transferencias**





# Forzar envío de Ethers



# Selfdestruct

**Selfdestruct** es una función de Solidity que transfiere los fondos de un contrato a una cuenta y luego elimina un contrato de la red.

Se pensó como medida de seguridad de un contrato. Sin embargo, su uso no es recomendado.

# Selfdestruct

Esta llamada, por ser de emergencia transfiere sin importar que la cuenta que reciba pueda ser un contrato con funciones definidas. Es decir, ignora la existencia de **receive** y **fallback**.





# Reentrancy simple

# Reentrancy

Es uno de los ataques más conocidos y peligrosos, ya que puede quitarle todos los fondos a una cuenta. Consiste en llamar recursivamente a una misma función hasta que no haya más gas o fondos que transferir.

**¿Cómo es posible?**





# Reentrancy cruzado

## **Reentrancy cruzado**

Es similar al *Reentrancy simple*, con la diferencia que en vez de llamar a la misma función, realiza la llamada sobre otra función del mismo contrato.





# **Denegación de servicio**





# Denegación por reversión



# Denegación de servicio

Es un ataque popular por el cual se interrumpe el acceso a un servidor, sitio o aplicación web.

En el caso de un contrato, su denegación es la interrupción del funcionamiento normal del mismo.

## **Por reversión**

En este caso, el contrato interrumpe su normal funcionamiento, ya que se produce una reversión cuando se lo quiere ejecutar.





# Denegación por límite de gas

## **Por límite de gas**

Una función puede ser interrumpida por límite de gas aun cuando no sea víctima de un ataque.

En caso de realizar varias transferencias en una misma llamada debemos contemplar que algunas pueden completarse y otras no.



# **Por bloque completo**

Una variante al límite de gas es cuando se completan los bloques con transacciones de alta prioridad evitando que una transacción específica se ejecute.

Esto suele funcionar cuando existen demoras de tiempo intencionales y la recompensa es alta.



# Desafío





# Analizar el siguiente contrato

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.7;


contract Desafio{
    uint private pin;
    mapping(address => uint) balances;

    constructor(uint ownerPin) {
        pin = ownerPin;
    }

    function mint(uint ownerPin, uint amount) public {
        require(pin == ownerPin, "El pin no es correcto.");
        balances[msg.sender] += amount;
    }

    function depositar() public payable {
        balances[msg.sender] += msg.value;
    }

    function retirar() public {
        require(balances[msg.sender] > 0);
        msg.sender.call{value:balances[msg.sender]}("");
        balances[msg.sender] = 0;
    }
}
```



```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.7;

contract Desafio{
    uint private pin;
    mapping(address => uint) balances;

    constructor(uint ownerPin) {
        pin = ownerPin;
    }

    function mint(uint ownerPin, uint amount) public {
        require(pin == ownerPin, "El pin no es correcto.");
        balances[msg.sender] += amount;
    }

    function depositar() public payable {
        balances[msg.sender] += msg.value;
    }

    function retirar() public {
        require(balances[msg.sender] > 0);
        msg.sender.call{value:balances[msg.sender]}("");
        balances[msg.sender] = 0;
    }
}
```





**Continúa aprendiendo**



# Sebastián Pérez



@sebaleoperez  
@blockacademycl