Informe: Implementación de Contrato Inteligente de Escrow

Curso: IC3272 - Criptomonedas y Contratos Inteligentes
Alumno: Emmanuel Norambuena
Fecha: 2 de Diciembre, 2025

1. Descripción del Caso de Uso

El objetivo de esta tarea fue desarrollar un contrato inteligente en Solidity que resuelva un problema de confianza en transacciones digitales peer-to-peer (P2P). Para ello, se implementó un sistema de **Escrow (Depósito en Garantía)**.

En un escenario de compraventa tradicional sin intermediarios, existe un riesgo inherente: el comprador teme pagar y no recibir el producto, y el vendedor teme enviar el producto y no recibir el pago. Este contrato actúa como un intermediario inmutable y descentralizado que retiene los fondos y solo los libera cuando se cumplen las condiciones preestablecidas, eliminando la necesidad de un tercero de confianza tradicional (como un banco o notario).

2. Arquitectura y Lógica del Contrato

El contrato `EscrowSeguro` funciona como una **Máquina de Estados Finitos**, garantizando que las acciones solo ocurran en el orden lógico correcto.

Roles

Se definen tres actores mediante direcciones (`address`):

- **Comprador:** Quien deposita los fondos.
- **Vendedor:** El beneficiario final de los fondos.
- **Árbitro:** Un tercero (en este caso, el desplegador) con capacidad de resolver disputas si el flujo normal falla.

Estados

Se utiliza un `enum` para gestionar el ciclo de vida del acuerdo: `CREADO` -> `PAGADO` -> `ENTREGADO` (o `REEMBOLSADO`).

Flujo Principal

1. **Instanciación:** Se despliega el contrato definiendo las direcciones del vendedor y el comprador.
2. **Depósito:** El comprador invoca la función `depositar()`. El contrato valida que el monto enviado sea mayor a 0 y cambia el estado a `PAGADO`. Los fondos quedan custodiados en la dirección del contrato.
3. **Liberación:** Una vez recibido el servicio/producto, el comprador llama a `confirmarEntrega()`. El contrato transfiere los ETH al vendedor y cierra el ciclo.

3. Análisis de Seguridad y Mejoras Implementadas

Para cumplir con el requisito de analizar la solución y garantizar la seguridad de los fondos, se implementaron los siguientes patrones de diseño y mitigaciones de vulnerabilidades:

1. **Protección contra Reentrancy:** Se implementó un modificador personalizado `noReentrancy` (actuando como un *mutex* o semáforo). Esto evita que contratos maliciosos llamen recursivamente a la función de retiro antes de que se actualice el saldo, protegiendo los fondos contra ataques comunes en DeFi.
2. **Uso de `.call` (Patrón Withdrawal):** Se utilizó la función de bajo nivel `.call{value: ...}("")` en lugar de `.transfer()`. Esta es la práctica recomendada actualmente en Solidity, ya que `transfer` impone un límite de gas fijo (2300) que puede hacer fallar la transacción si la billetera del vendedor es un contrato inteligente complejo (Smart Wallet).
3. **Mecanismo de Arbitraje (Fail-safe):** Para evitar el bloqueo permanente de fondos (*ETH Lock*) en caso de que el vendedor desaparezca o no cumpla, se incluyó la función `reembolsar()`. Esta función solo puede ser ejecutada por el `arbitro` y devuelve los fondos al comprador, garantizando la reversibilidad en casos de disputa.

4. Despliegue

El contrato fue compilado y desplegado exitosamente en la red de prueba (Testnet) de Ethereum requerida para la entrega.

- **Network:** Ethereum Sepolia Testnet
- **Dirección del Contrato:** 0x138944a021e24BCa53583782874268d34e9a6451
- **Compiler Version:** 0.8.26

5. Referencias

- Solidity Documentation. (n.d.). *Smart Contract Security Best Practices*. Recuperado de https://docs.soliditylang.org/
- OpenZeppelin. (n.d.). *ReentrancyGuard Source Code*. Recuperado de https://github.com/OpenZeppelin/
- Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum*. O'Reilly Media.