DEEP SECURITY

# ETHICAL HACKING SMART CONTRACTS

(2023)

# Indice [Septiembre]

Introducción [3]

¿La seguridad de los smart contracts debe ser evaluada? [3]

¿Por qué se siguen reportando incidentes en los smart contracts? [4]

Entendiendo el ecosistema del smart contract (blockchain) [5]

Las vulnerabilidades más comunes en los smart contracts [7]

Comentarios finales [18]

Referencias [19]

## METODOLOGÍA

**DeepSecurity** realizó una evaluación de vulnerabilidades que afectan a un Smart Contract mediante pruebas de concepto representativas de cada una de estas vulnerabilidades identificadas en el artículo.

El presente artículo menciona 5 vulnerabilidades más comunes donde las pruebas realizadas permitieron comprender y demostrar cómo las vulnerabilidades pueden ser explotadas en la práctica y qué impacto podrían tener en la seguridad del Smart Contract y su funcionalidad.

DEEPSECURITY

## INTRODUCCIÓN [3]

Esta introducción reúne a un equipo de consultores que poseen una amplia experiencia en áreas como pruebas de penetración, auditoría de código fuente y desarrollo de exploits, quienes realizaron el análisis sobre las vulnerabilidades más comunes asociadas con incidentes de seguridad a plataformas basadas en **"smart contracts"**.

En un momento en el que la tecnología de blockchain está en constante evolución, los profesionales de la ciberseguridad se enfrentan a nuevos desafíos en el ámbito de la tecnología de criptomonedas y ecosistemas relacionados.

A partir de nuestra experiencia colectiva, tenemos como objetivo comentar las complejidades de las evaluaciones a **smart contracts** y dar a conocer sobre los riesgos potenciales que plantean.

## ¿LA SEGURIDAD DE LOS SMART CONTRACTS DEBE SER EVALUADA?[3]

En estos tiempos los **smart contracts** manejan grandes cantidades de divisas tales como criptomonedas, inversiones o activos financieros descentralizados (DeFi - Decentralized Finance). Esto hace que actualmente sea un mercado atractivo a ciberdelincuentes quienes prestan atención en ellos, dado que pueden existir vulnerabilidades dentro del código del contrato que son explotadas por los atacantes para robar o manipular cientos o miles de millones de dólares como se ha reportado en los últimos incidentes.

Alguna de estas acciones por parte de los atacantes, puede ocasionar grandes pérdidas financieras a organizaciones, tales como:

1. Pérdidas en el ecosistema DEFI - Pérdida de US$ 60 millones
2. Ataque a Solana 2022 - Ocho mil billeteras vaciadas y US$ 8 millones robados
3. Hackeo a plataforma Nomad - Pérdida de US$ 190 millones
4. Plataforma de criptomonedas Wormhole pirateada después de un error en GitHub - $ 325 millones.

## ¿POR QUÉ SE SIGUEN REPORTANDO INCIDENTES EN LOS SMART CONTRACTS? [4]

Hay varias razones por las que los **smart contracts** siguen siendo un objetivo para un actor de amenaza:

1. **Falta de experiencia**: el desarrollo de contratos inteligentes requiere un alto nivel de experiencia tanto en tecnología blockchain como en lenguajes de programación como Solidity, Vyper, Ride, entre otros. Sin embargo, es posible que muchos desarrolladores no tengan las habilidades y la experiencia necesarias para escribir código seguro. Esto puede generar vulnerabilidades y errores que los atacantes pueden aprovechar.
2. **Código complejo**: los contratos inteligentes pueden ser muy complejos, especialmente cuando involucran múltiples funciones e interacciones con otros contratos. Esta complejidad puede dificultar la identificación y el tratamiento de posibles vulnerabilidades.
3. **Falta de estandarización**: actualmente no existen estándares ampliamente aceptados para el desarrollo de contratos inteligentes, lo que significa que cada desarrollador o equipo puede usar su propio enfoque de codificación. Esta falta de estandarización puede dificultar que los desarrolladores aprendan de los errores de los demás y mejoren su código.
4. **Error humano**: Incluso los desarrolladores experimentados pueden cometer errores al escribir código, especialmente cuando están trabajando en un proyecto complejo. Un solo error en un smart contract puede generar vulnerabilidades que los atacantes pueden aprovechar.
5. **Falta de Auditorías especializadas**: las pruebas manuales exhaustivas son esenciales para identificar y abordar las vulnerabilidades en los contratos inteligentes. Sin embargo, es posible que muchas empresas auditoras no lleven a cabo las pruebas adecuadas, dejando su código abierto a posibles ataques.

Para abordar estos problemas, los desarrolladores deben priorizar la seguridad en el desarrollo de sus contratos inteligentes, buscar asesoramiento y orientación de expertos y seguir las mejores prácticas para la codificación y las pruebas seguras. Además, la comunidad de desarrollo en su conjunto puede trabajar hacia la estandarización y la mejora de las prácticas de seguridad para reducir el riesgo de ataques y vulnerabilidades.

## ENTENDIENDO EL ECOSISTEMA DEL SMART CONTRACT (BLOCKCHAIN)[5]

El ecosistema de contratos inteligentes se refiere al entorno colectivo, las tecnologías y los componentes relacionados con la creación, implementación y utilización de **smart contracts** en sistemas basados en blockchain.

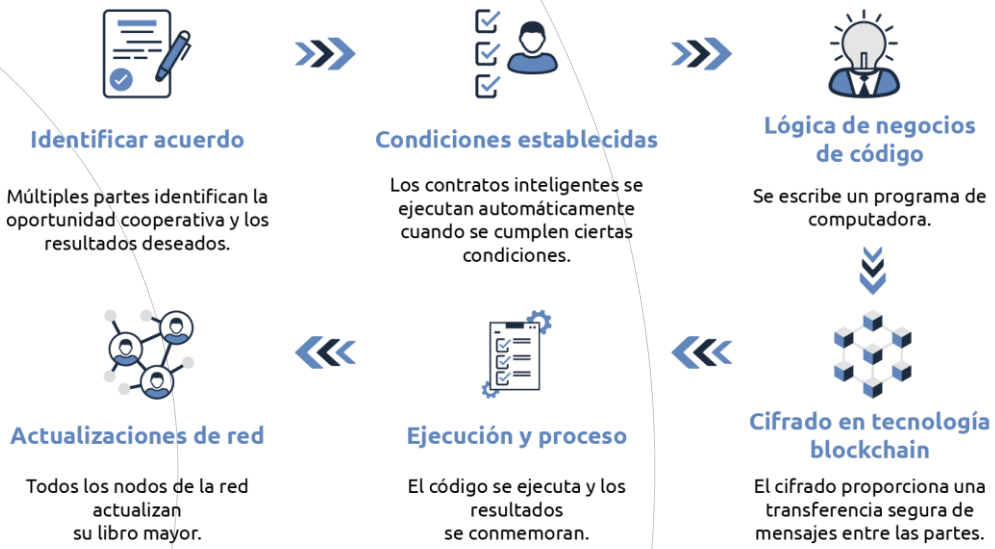


Imagen 1: Diagrama de Flujo Regular

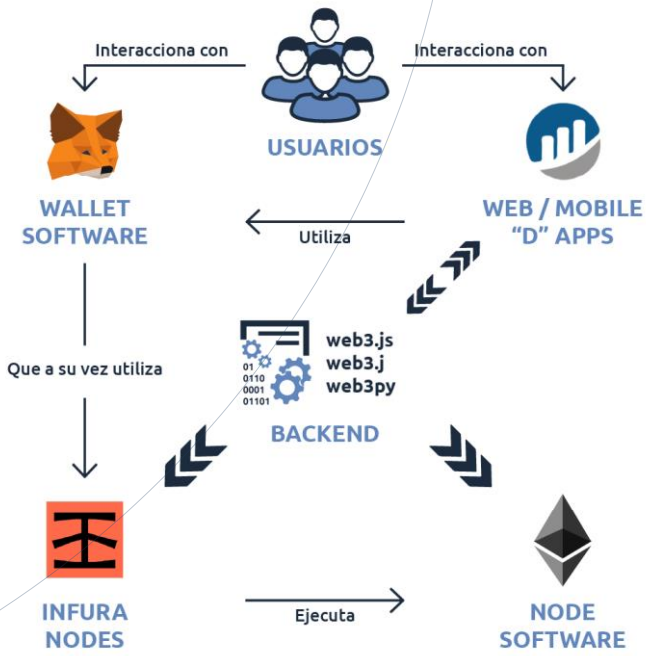


Imagen 2: Diagrama de Flujo de Análisis

## Para desplegar “Smart Contracts” en el ecosistema de la blockchain, se conectó nuestra billetera(metamask) con un nodo de “Infura” usando la biblioteca web3.py de python.

```
# print(compiled_sol)

with open("compiled_code.json", "w") as file:
    json.dump(compiled_sol, file)

# bytecode
# ABI

bytecode = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"][
    "bytecode"
]["object"]

abi = json.loads(
    compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["metadata"]
)["output"]["abi"]


# w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
w3 = Web3(
    Web3.HTTPProvider("https://sepolia.infura.io/v3/5862e375077e4e7aae5af01a5f48f93b")
)
chain_id = 11155111
# my_address = "0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1"
my_address = "0x358356edc9d842eFE062595f0edf18a96422C6dc"
private_key = os.getenv("PRIVATE_KEY")

SimpleStorage = w3.eth.contract(abi=abi, bytecode=bytecode)
```

Imagen 3: Despliegue del Smart Contract en la Blockchain

Después de desplegar los **smart contracts** y ya teniendo los contratos en la blockchain es posible revisar las interacciones que tiene este contrato con otros, como realizar el seguimiento de una transacción y su historial, saber la información de un token, verificar códigos fuentes y códigos de bytes, monitorear las transacciones en la red de Ethereum, observar gráficos de precio y datos del mercado, y rastrear el rendimiento de varios token ERC-20 que son un estándar técnico utilizado para implementar tokens fungibles en la blockchain de Ethereum. Los tokens fungibles son intercambiables e idénticos entre sí, lo que significa que cada token tiene el mismo valor y se puede intercambiar uno a uno.

Para realizar el seguimiento de nuestras transacciones en la blockchain puedes visitar la siguiente dirección: https://sepolia.etherscan.io/address/0x358356edc9d842eFE062595f0edf18a96422C6dc

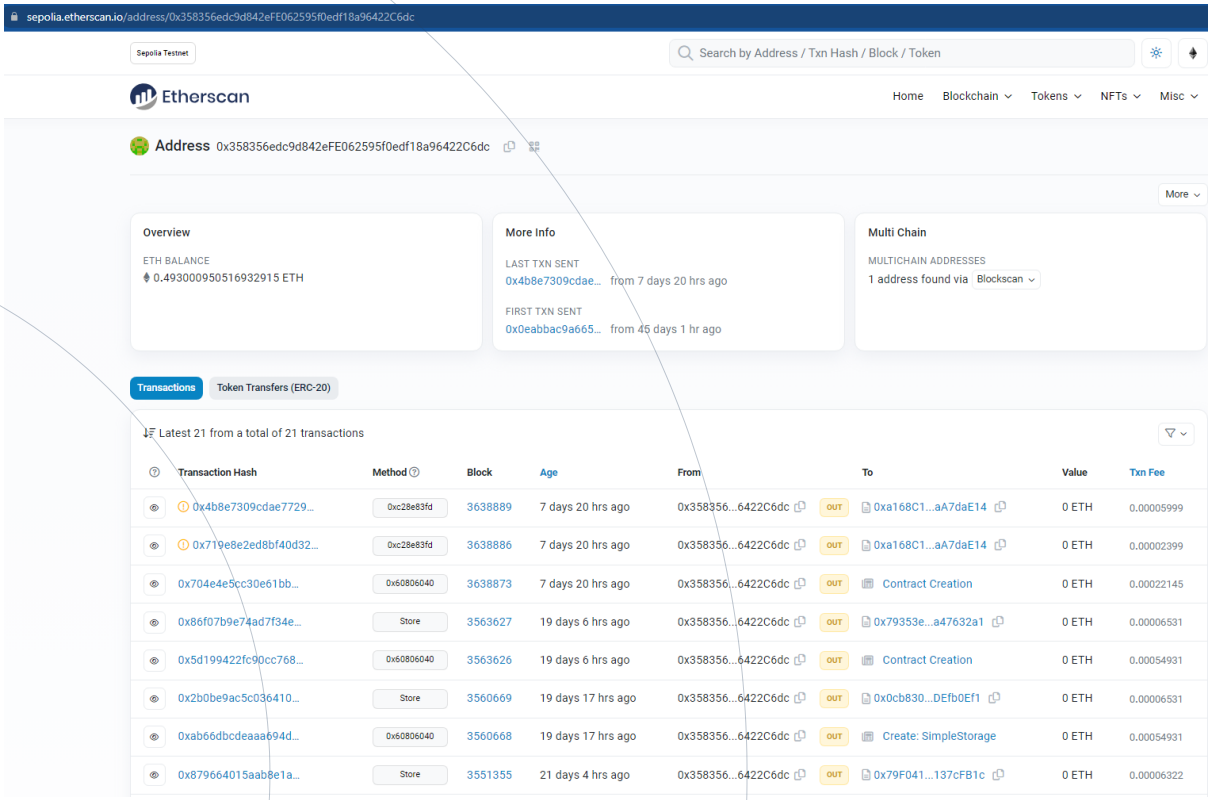


Imagen 4: Historial de nuestro contrato desplegado en la blockchain

## LAS VULNERABILIDADES MÁS COMUNES EN LOS SMART CONTRACTS [7]

Los **smart contracts** han ganado una inmensa popularidad debido a su capacidad para automatizar y hacer cumplir los acuerdos sin intermediarios. Sin embargo, no son inmunes a las vulnerabilidades que pueden conducir a graves riesgos financieros y de seguridad. En este artículo, exploraremos las vulnerabilidades más comunes que se encuentran en **smart contracts**, destacaremos la importancia de comprender la lógica de las vulnerabilidades y desarrollaremos ejemplos de **smart contracts** vulnerables.

### 1. Integer over- and underflows.

Si hay un límite para el tamaño de los números almacenados en las computadoras, ¿qué sucede cuando cruzamos este límite?

En Ethereum, cada ranura entera sin firmar tiene un espacio de almacenamiento de 32 bytes o 256 bits. Digamos que desea realizar una suma aritmética entre 2 enteros legítimos, pero sin signo:

- A = 0x00…001 (1)

- B = 0xFF…FFF ($2^{256}$ - 1, valor máximo sin signo)
- C = A + B = 0x00…001+ 0xFF…FFF = 1 + ($2^{256}$ - 1) =
- $2^{256}$ = 0x **1** 00…000

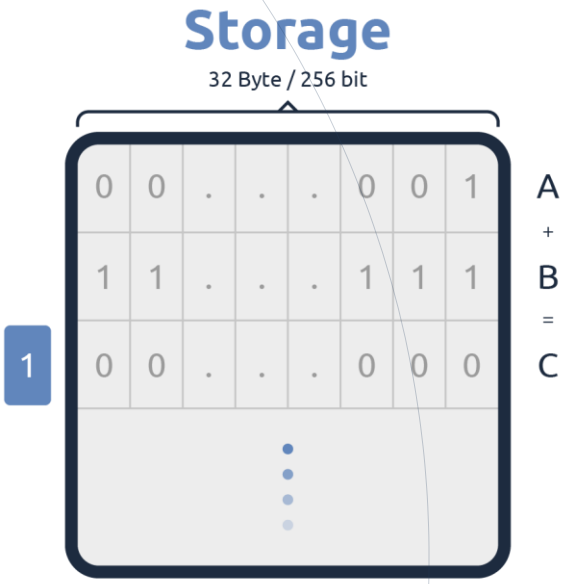


Imagen 5 : Lógica de la vulnerabilidad Integer over- and underflows

El resultado de esta suma aritmética es un número mayor que el máximo entero posible.

La variable usada representa el número positivo 1 seguido de 256 ceros (b100…000), que en total tiene una longitud de 257 bits. Pero la ranura del valor entero en el almacenamiento solo puede tener 256 bits. Por lo tanto, solo los 256 (bits más a la derecha) se almacenan y todo lo demás se ignora.

Como resultado, el valor que realmente se almacena es 0x00…000 (0), esto es un desbordamiento de enteros.

```
contract TimeLock {
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = block.timestamp + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0, "Insufficient funds");
        require(block.timestamp > lockTime[msg.sender], "Lock time not expired");

        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Imagen 6: Contrato vulnerable a Interoverflow.

El incidente de ciberseguridad que afectó a DAO, también conocido como ataque DAO o exploit DAO, ocurrió en el 2016 y fue uno de los eventos más significativos en la historia de las criptomonedas y la tecnología blockchain. La DAO era una organización autónoma descentralizada construida sobre la blockchain de Ethereum, diseñada para operar como un fondo de capital de riesgo regido por smart contracts.

La vulnerabilidad en el código del smart contract de DAO permitió a un atacante manipular el sistema de contabilidad interno. Al explotar un error de integer underflow, el atacante pudo solicitar fondos repetidamente del DAO sin actualizar el saldo de su cuenta. Esto les permitió drenar una cantidad significativa de Ether de la DAO, lo que generó pérdidas financieras sustanciales en este caso de $60 millones.

## 2. Re-Entrancy (re-ingreso)

Los contratos maliciosos llaman repetidamente a una función en otro contrato antes de que finalice la primera llamada de función, lo que podría causar un comportamiento inesperado y robar fondos.

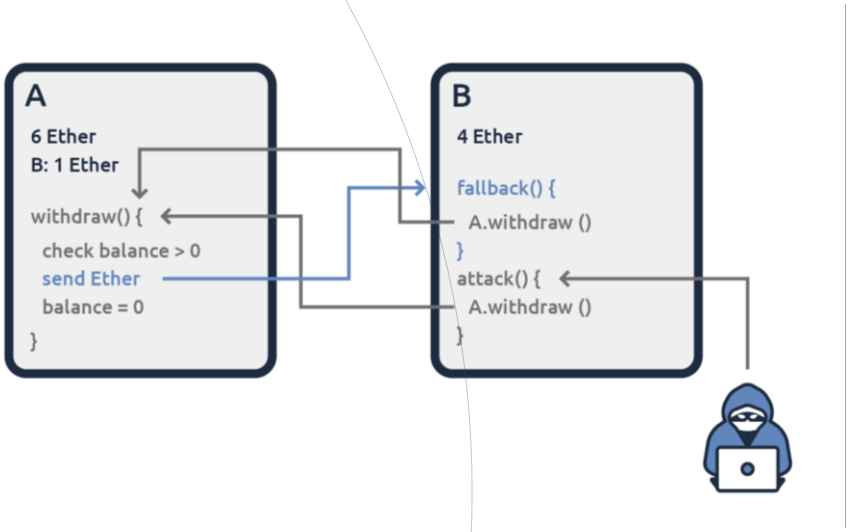


Imagen 7: Lógica de la vulnerabilidad Re-Entrancy

1. El "contrato B" pediría un retiro del "contrato A" llamando la función ataque.

2. "Contrato A" llama a la función de retiro y envía 1 ether a "contacto B" después de confirmar B tiene más de 0 ether en el contrato

3. La transferencia del "Contrato A" al "Contrato B" ha activado una función de respaldo.

4. La función de retroceso le pediría otra retirada.

5. La transferencia del "Contrato A" al "Contrato B" ha activado una función de respaldo nuevamente

6. La función de retroceso le pedirá otra vez que se retire: el proceso se repite.

Observe que el código de la función *call()* puede ejecutar una función en otro contrato. Luego de esto, se podría llamar a esta función, lo que resultaría en la ejecución de un retiro parcial (withdraw) varias veces antes de que se reduzca el saldo.

```
contract Reentrance {


  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] = balances[_to]+msg.value;
  }

  function balanceOf(address _who) public view returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      (bool result,) = msg.sender.call{value:_amount}("");
      if(result) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  receive() external payable {}
}
```

Imagen 8: Contrato vulnerable a Re-entrancy (re-ingreso).

Para evitar ataques de Re-entrancy (re-ingreso), generalmente se recomienda seguir el patrón de **"comprobaciones-efectos-interacciones"** en Solidity. En el siguiente código modificado, agregamos una nueva asignación *"locked"* para realizar un seguimiento de si un usuario tiene una transacción pendiente o no. La función withdraw() ahora verifica si el usuario tiene saldo suficiente y si no tiene transacciones pendientes antes de proceder con el retiro. Si las comprobaciones tienen éxito, bloquea la cuenta del usuario, deduce el monto solicitado de su saldo, envía el monto solicitado y luego desbloquea la cuenta. Si alguna de estas operaciones falla, la función se revertirá y devolverá un mensaje de error. lo que significa realizar todas las comprobaciones y modificar el estado del contrato (es decir, los efectos) antes de interactuar con contratos externos o direcciones de usuario.

Aquí podemos visualizar un ejemplo de cómo modificar la función 'withdraw' para evitar ataques de re-ingreso (Re-Entrancy):

```
contract Reentrance {

  mapping(address => uint) public balances;
  mapping(address => bool) public locked;

  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }

  function balanceOf(address _who) public view returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    require(balances[msg.sender] >= _amount, "Insufficient balance");
    require(!locked[msg.sender], "Your previous transaction is still pending");

    locked[msg.sender] = true;
    balances[msg.sender] -= _amount;

    (bool result,) = msg.sender.call{value:_amount}("");
    require(result, "Withdrawal failed");

    locked[msg.sender] = false;
  }

  receive() external payable {}
}
```

Imagen 9: Contrato modificado, no es vulnerable (Re-Entrancy).

Uno de los principales incidentes de seguridad en DeFI fue el protagonizado por "Sturdy Finance", un proyecto de finanzas descentralizadas (DeFi), se vio obligado a detener sus operaciones de mercado debido a un exploit de $800,000 que estaba vinculado a una fuente de datos de precios defectuosa transmitida a la blockchain. El incidente ocurrió el 12 de junio de 2023 y subrayó los riesgos asociados a confiar en fuentes de datos externas para determinar los precios de los tokens en los protocolos DeFi.

En este exploit en particular, el origen de datos respecto a precios utilizada por "Sturdy Finance" proporcionó información de precios inexacta, que fue manipulada por los atacantes. Al explotar esta vulnerabilidad, los atacantes pudieron manipular los precios de los tokens dentro del protocolo, creando oportunidades de manipular arbitrariamente los precios y beneficiándose a expensas de otros usuarios.

## 3. Front Running

La ejecución anticipada de esta vulnerabilidad puede ocurrir cuando un usuario envía una transacción a un **smart contract**, y un actor malicioso monitorea la blockchain (mempool) para detectar la transacción antes de que se confirme.

Luego, el atacante envía su propia transacción con un precio de gas más alto, lo que le permite ejecutar la misma antes de la original.

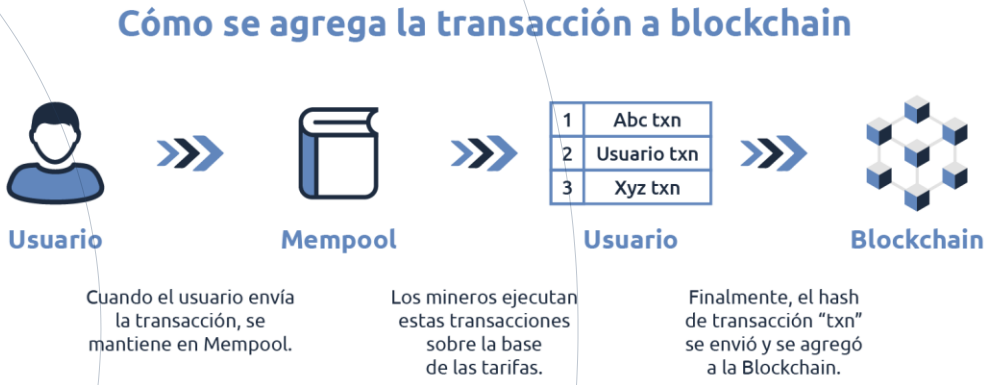


Imagen 10: Lógica de una transacción común.

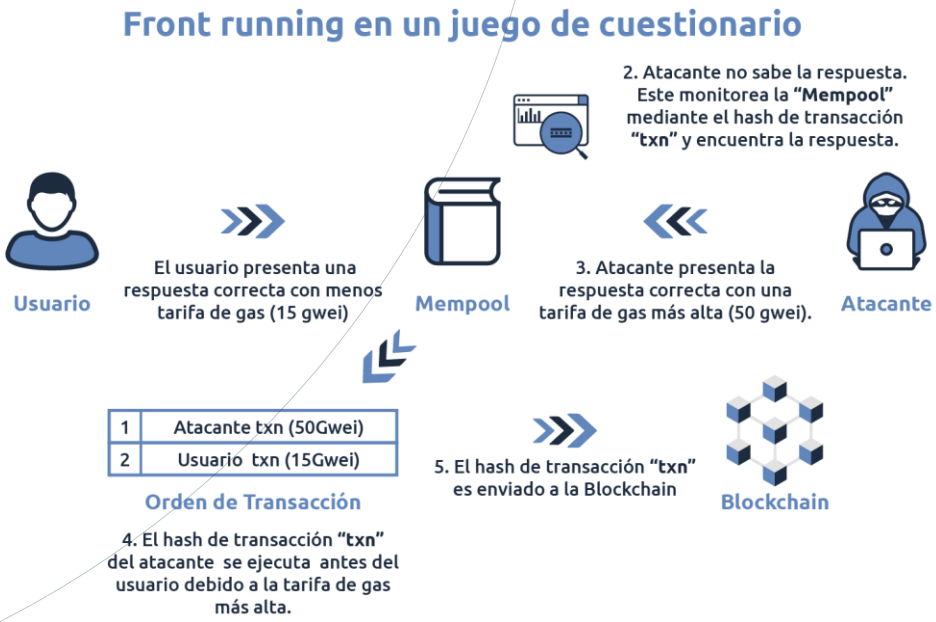


Imagen 11: Lógica de la vulnerabilidad FRONT RUNNING

```
contract FindThisHash {
    bytes32 public constant hash =
        0x686f6c6100000000000000000000000000000000000000000000000000000000;

    constructor() payable {}    288907 gas 288600 gas

    function solve(string memory solution) public {    infinite gas
        require(hash == keccak256(abi.encodePacked(solution)), "Incorrect answer");

        (bool sent, ) = msg.sender.call{value: 10 ether}("");
        require(sent, "Failed to send Ether");
    }
}
```

Imagen 12: Contrato vulnerable a FRONT RUNNING

Un gran ejemplo de esta vulnerabilidad es el ataque que se realizó a los primeros inversionistas de UNISWAP que buscaban ventajas en el mercado. Este caso trata acerca de que desde ocho direcciones se lograron robar $25,2 millones en activos mediante bots de front-running ejecutando la modalidad de ataque de “sandwich” que es una variante de Front-running.

El atacante supervisó la mempool o las transacciones pendientes en la red de Ethereum buscando grandes órdenes de compra o venta que puedan afectar significativamente el precio de un token en UNISWAP. Estas transacciones suelen ser enviadas por otros comerciantes. El atacante envía rápidamente sus propias transacciones antes y después de la transacción del comerciante para aprovechar el movimiento de precios causado por la operación de este. El objetivo es comprar barato y vender caro para beneficiarse de la discrepancia de precio, con ello el atacante se asegura de que sus transacciones se incluyan en el bloque antes y después de la transacción de destino. De esta forma, pueden manipular el precio de mercado y maximizar sus ganancias.

## 4. Denegación de Servicio (DoS)

Es el intento de interrumpir u obstaculizar el funcionamiento normal del contrato, saturando sus recursos o provocando un consumo excesivo de gas.

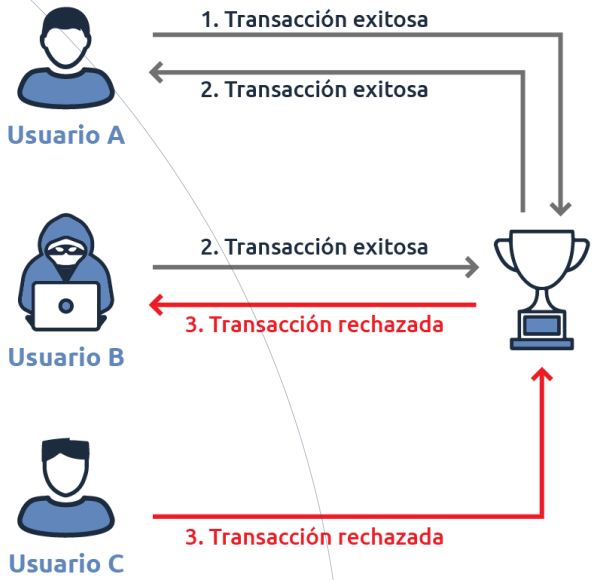


Imagen 13: Lógica de la vulnerabilidad DOS

Iniciamos desplegando el **smart contract** "Subasta", después:

1. **"Usuario A"** se convierte en el "mejor postor" enviando 1 ether.

2. **"Usuario B"** se convierte en el "mejor postor" enviando 2 ether, entonces **"Usuario A"** recibe un reembolso de 1 ether.

3. **"Usuario C"** puja enviando 3 ether, pero el **"Usuario C"** no se convierte en el mejor postor, a pesar de enviar mayor cantidad de ether, esto se debe a que **"Usuario B"** crea una función alternativa que revierte todos los reembolsos.

4. El mejor postor es el "**Usuario B"** y nadie puede convertirse en el nuevo mejor postor.

```
contract Auction {
    address public currentLeader; //Current bidder
    uint256 public highestBid; // The highest bid

    function bid() public payable {
        require(msg.value > highestBid);
        // Reembolsar el líder anterior, si falla, revertir
        require(currentLeader.send(highestBid));
        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

Imagen 14: Contrato vulnerable a DOS

Otro de los ataques más resaltantes fue realizado a Solana, quién después de sufrir una interrupción por denegación de servicio perdió el 15% del valor en las últimas 24 horas el 14 de septiembre de 2021 logrando afectar su precio en el mercado debido a la caída en el sistema pasando de cotizarse $175 a $145, siendo la noticia de este ataque la detonante de que los precios cayeran debido a la incertidumbre y preocupación.

La denegación de servicio se dio al gran aumento de la carga de transacciones que fueron 400000 por segundo lo que causó la saturación de la red, de esta manera, cuando no se pudo estabilizar, se optó por coordinar un reinicio de la red. Así mismo, el día martes 14 de septiembre, una entidad desconocida intentó atacar Ethreum sin éxito.

## 5. Signature Malleability (Maleabilidad de la firma)

Ocurre cuando se utilizan incorrectamente las firmas **ECDSA (Algoritmo de firma digital de curva elíptica)**. La vulnerabilidad permite que un atacante cambie ligeramente la firma sin validarla en sí. Esto sucede a menudo cuando **un smart contact** no valida las firmas correctamente, lo que permite a los atacantes modificarlas y potencialmente eludir las medidas de seguridad.

Una curva elíptica consta de todos los puntos que satisfacen una ecuación de la forma:

$$y^2 = x^3 + ax + b$$

Las curvas siempre son simétricas con respecto al eje x.

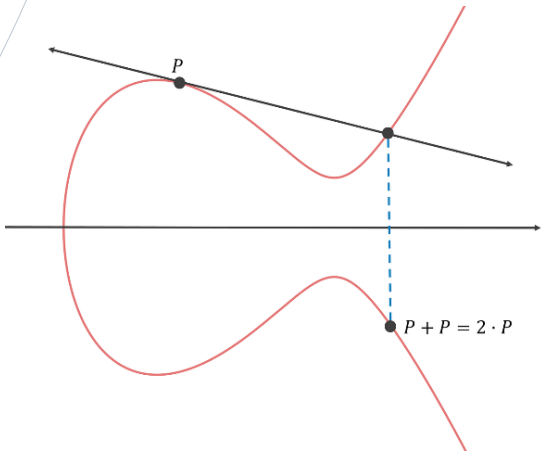


Imagen 15: Curva elíptica

Las firmas ECDSA consisten en un par de números, (r, s), de orden entero n. Como resultado de la simetría del eje x, si (r, s) es una firma válida, entonces también lo es (r, −s mod n).

## Donde:

**r:** es un valor derivado del punto de la curva elíptica (x, y) generado durante el proceso de firma. El r valor es la coordenada x de ese punto.

**s:** es un valor calculado durante el proceso de firma utilizando la clave privada, el hash del mensaje y r. El valor s está destinado a demostrar que el firmante tiene conocimiento de la clave privada

**v:** es el identificador de recuperación, que se utiliza para recuperar la clave pública correcta (dirección) de la firma.

Es posible calcular esta firma complementaria sin conocer la clave privada utilizada, lo que le da al atacante la capacidad de producir una segunda firma válida.

```
function performTransaction(    infinite gas
    bytes memory signature,
    bytes32 messageHash
)
    public
    returns (bool)
{
    require(!messageUsed[messageHash], "Mensaje ya utilizado");

    // Verifique la firma y recupere la dirección del firmante
    address signer = recoverSigner(signature, messageHash);
```

Imagen 16: Contrato vulnerable a Signature Malleability

En el ejemplo anterior, podemos ver que “messageHash” se guarda en una asignación “messageUsed” después de la ejecución y se valida para que no exista en esa asignación antes de la ejecución. El problema con esto es que, si “messageHash” puede modificar manteniendo la validez, un atacante puede repetir la transacción.

Para evitar este problema, es imperativo reconocer que para validar que una firma no se reutilice no solo basta con hacer cumplir que esta sea válida.

Mt. Gox ha experimentado pérdidas sustanciales. Este ataque aprovechó una vulnerabilidad conocida como Signature Malleability, que permitió a los

atacantes manipular el identificador único de transacción (TXID) de las transacciones de Bitcoin. Al alterar el TXID, los atacantes podían hacer que pareciera que una transacción no se había procesado, lo que les permitía solicitar retiros repetidamente de Mt. Gox sin que el intercambio detectara la duplicación.

Este ataque de Signature Malleability en Mt. Gox resultó en pérdidas de $600 millones en Bitcoins lo que socavó la confianza de los usuarios en el intercambio. Desde entonces, el incidente ha servido como una advertencia para la industria, enfatizando la necesidad de mejorar las prácticas de seguridad y la implementación de salvaguardas para prevenir este tipo de ataques en el futuro.

## COMENTARIOS FINALES [18]

En conclusión, mejorar el conocimiento de las finanzas descentralizadas (DeFi) por parte de las empresas auditoras y realizar evaluaciones exhaustivas de seguridad como ethical hacking o bug-bounty son cruciales para identificar y mitigar vulnerabilidades específicas en contratos inteligentes y protocolos DeFi. Dado el rápido crecimiento y la complejidad del ecosistema DeFi, es esencial que los profesionales de la ciberseguridad se familiaricen con los aspectos únicos de la tecnología blockchain y las criptomonedas.

Es importante reconocer que muchos ataques a las plataformas y protocolos DeFi no se originan a partir de vulnerabilidades dentro de las propias tecnologías de cadena de bloques subyacentes. Si bien la tecnología blockchain proporciona una base segura e inmutable, a menudo son los componentes externos, como los **smart contracts**, las aplicaciones descentralizadas (dApps) y las interacciones de los usuarios, donde se explotan las vulnerabilidades.

Para abordar estas preocupaciones, han surgido varias iniciativas y mejores prácticas para mejorar la seguridad de estos como, por ejemplo:

- Auditorías de código fuente.
- Programas de recompensas por errores (bugbounty) especializados
- Prácticas de desarrollo seguro.

Si deseas implementar la seguridad en tus **smart contracts** en gestión o estás en alguna de las situaciones de la información brindada, no dudes en contactarnos comercial@deepsecurity.pe .

# REFERENCIAS [19]


### SolidityScan

Aquí hay algunos artículos que pueden brindarle una comprensión más profunda de las vulnerabilidades en los contratos inteligentes:

1. https://blog.solidityscan.com/
2. https://medium.com/@knownsec404team/ethereum-smart-contract-audit-checklist-ba9d1159b901
3. https://www.infuy.com/blog/preventing-denial-of-service-attacks-in-solidity/
4. https://blog.finxter.com/denial-of-service-dos-attack-on-smart-contracts/
5. https://solidity-by-example.org/
6. https://github.com/kadenzipfel/smart-contract-vulnerabilities/tree/master/vulnerabilities
7. https://swcregistry.io/
8. https://programtheblockchain.com/posts/2018/02/17/signing-and-verifying-messages-in-ethereum/
9. https://github.com/obheda12/Solidity-Security-Compendium/blob/main/days/day12.md
10. https://eklitzke.org/bitcoin-transaction-malleability
11. https://www.getsecureworld.com/blog/smart-contract-gas-griefing-attack-the-hidden-danger/
12. https://redfoxsec.com/blog/integer-overflow-in-smart-contract/
13. https://assets.ctfassets.net/hfgyig42jimx/2l90zr21hmUG2aL7eK02Cl/ccb091f0c5247abaed4984bc3c286782/Attacks_and_Exploits_in_DeFi.pdf
14. https://quillaudits.medium.com/front-running-and-sandwich-attack-explained-quillaudits-de1e8ff3356d

### Tools

Existen varios escáneres y herramientas populares que se utilizan para encontrar vulnerabilidades en los contratos inteligentes. Estos son algunos de los más utilizados:

1. https://github.com/ConsenSys/mythril
2. https://www.alchemy.com/dapps/slither
3. https://github.com/eth-sri/securify2
4. https://github.com/ethereum/oyente
5. https://github.com/crytic/echidna

### Casos reales

Puedes revisar los casos más importantes causados por las vulnerabilidades mencionadas en este artículo.

1. https://www.coindesk.com/consensus-magazine/2023/05/09/coindesk-turns-10-how-the-dao-hack-changed-ethereum-and-crypto/
2. https://www.financialexpress.com/business/blockchain-sturdy-finance-sustains-800000-worth-losses-through-hack-3123277/
3. https://u.today/mev-front-running-bots-exploited-252-million-stolen-by-sandwich-attackers

DEEP SECURITY

**Colaboradores DeepSecurity**

**Investigación Técnica:**
Mauricio Urizar y Percy Jayo

**Creativo:**
Deysi Espíritu

Sobre Deep Security
Deep Security publica artículos originales, informes y publicaciones periódicas que proporcionan información para las empresas, el sector público y sociedad en general. Nuestro objetivo es aprovechar la investigación y laexperiencia de toda nuestra organización de servicios profesionales para avanzar en el desarrollo sobre un amplio espectro de temas de interés para ejecutivos y líderes del gobierno.

Deep Security 100% peruana.