Università di Napoli Federico II - Dip. di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea in Ingegneria Informatica

Vulnerabilità di Underflow e Overflow in Smart Contract Solidity

Dimostrazione pratica e mitigazione con Remix e Metamask

Underflow e Overflow in Smart Contract Solidity

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 **overflow** e **underflow** rappresentano una classe di vulnerabilità critica nei sistemi informatici, particolarmente pericolosa nel contesto degli **smart contract**, dove l'immutabilità del codice impedisce *correzioni post-deployment*. Questa presentazione dimostra come questi attacchi possano compromettere la logica di business di un contratto per la creazione di Dynamic NFT e come mitigarli efficacemente.

Cos'è un Overflow/Underflow?

Overflow:

- Si verifica quando un'operazione aritmetica produce un risultato che *eccede* il valore massimo rappresentabile per un tipo di dato
- Per esempio, in un uint8 (range 0-255), aggiungere 1 a 255 produrrebbe 256, che non è rappresentabile in 8 bit
- Il risultato "riparte" da 0: 255 + 1 = 0 invece di 256

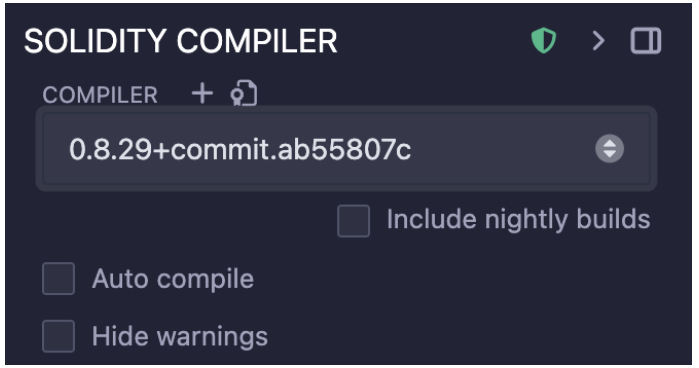
Underflow:

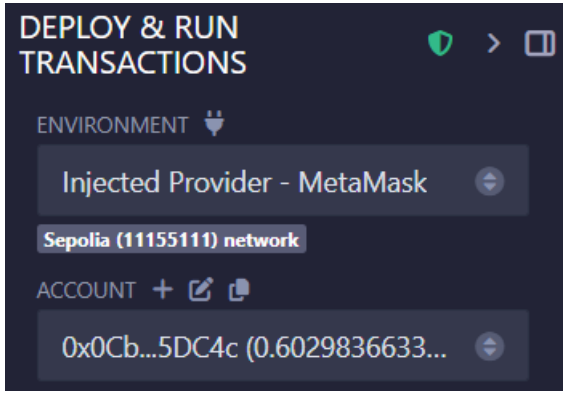
- Si verifica quando un'operazione aritmetica produce un risultato *inferiore* al valore minimo rappresentabile
- Per esempio, in un uint8, sottrarre 1 da 0 produrrebbe -1, che non è rappresentabile in un tipo unsigned
- Il risultato "riparte" dal valore massimo: 0 - 1 = 255 invece di -1

Osservazione: questi comportamenti derivano dalla rappresentazione binaria dei numeri. Quando si lavora con *numeri a dimensione fissa* (come uint8, uint256 in Solidity), il sistema non può gestire valori al di fuori del range previsto, causando un "**wrap around**" (riavvolgimento) ai valori estremi opposti.

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

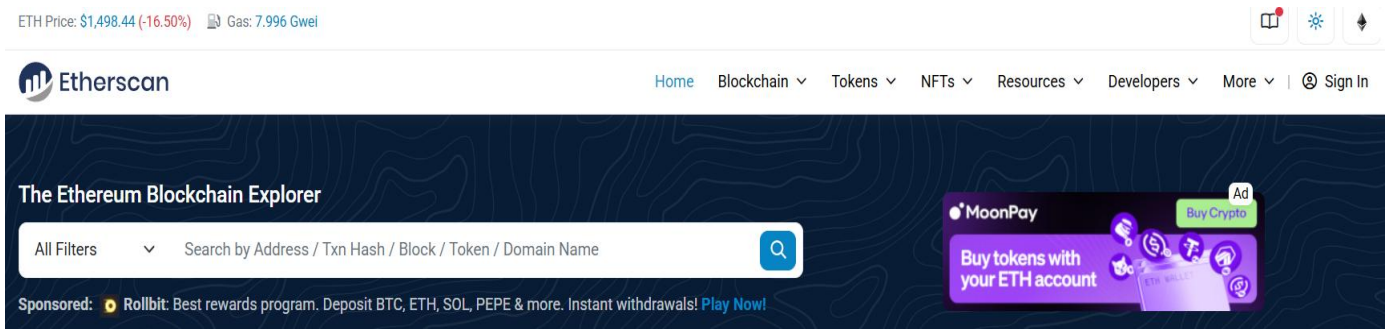




Setup dell'Ambiente (2/2)

- **MetaMask**: Wallet Ethereum per firmare le transazioni (funziona come **estensione del browser** o app mobile)
  - Configurato per connettersi a una *rete di test* (es: Sepolia)

- **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.

Il Contratto Vulnerabile

Struttura del contratto NFT con sistema di reputazione:

```
21
22      /// @notice Crea un NFT dinamico con reputazione iniziale
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          balances[newTokenId] = 0;
28          reputation[newTokenId] = 100;
29          tokenCounter += 1;
30          emit NFTCreated(newTokenId, tokenURI);
31          return newTokenId;
32      }
```

```
48      /// @notice Simula una ricompensa che può causare overflow
49      function reward(uint256 tokenId) public {    29402 gas
50          require(ownerOf(tokenId) == msg.sender, "Not your NFT");
51          unchecked {
52              reputation[tokenId] += 10; // Vulnerabile a overflow
53          }
54      }
55
56      /// @notice Simula una penalità che può causare underflow
57      function penalize(uint256 tokenId) public {    26959 gas
58          unchecked {
59              reputation[tokenId] -= 20; // Vulnerabile a underflow
60          }
61      }
```

Punti critici da evidenziare:

- Utilizzo di uint8 per la reputazione, limitando i valori tra 0 e 255
- Mancanza di controlli nelle funzioni *reward()* e *penalize()*: questa è stata ottenuta andando a **disattivare** il controllo implicito effettuato dal compilatore (*unchecked*)
- Operazioni aritmetiche non sicure (+=, -=)

Meccanismo di reputazione

A cosa può servire **compromettere** un **sistema di reputazione**:

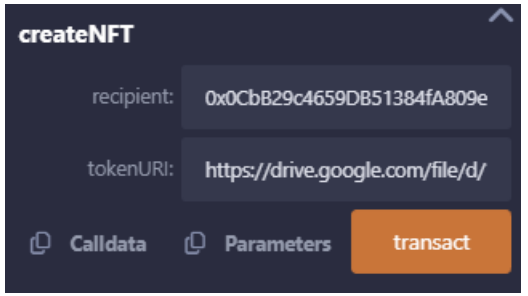
- avere una priorità maggiore nel depositare i fondi per il proprio NFT
- assegnare una priorità minore nel depositare i fondi agli NFT altrui
- avere una reputazione maggiore/minore in un sistema di voting (per esempio rispetto alle transazioni) può impattare su quali transazioni validare

N.B. utenti con reputazione più alta sono ritenuti più affidabili.

Creazione e Stato Iniziale (1/2)

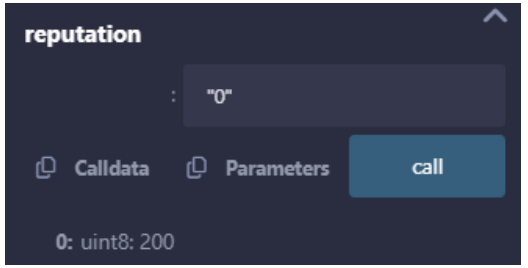
Processo di creazione NFT:

1. Chiamata alla funzione *createNFT()* tramite Remix
2. Firma della transazione con MetaMask
3. Attesa della conferma sulla blockchain



Analisi dello stato iniziale:

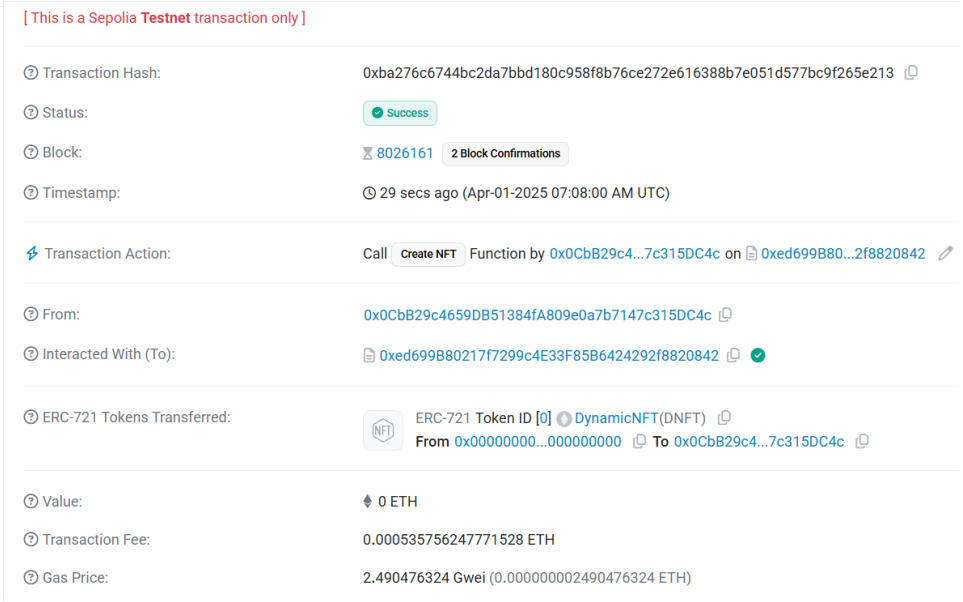
- Ogni nuovo NFT viene creato con una reputazione iniziale di 200
- Questo valore è stato scelto strategicamente per dimostrare l'overflow dopo poche operazioni di reward
- Il tokenId viene incrementato automaticamente quando si crea un NFT (partendo da 0): serve a specificare l'NFT sul quale stiamo operando



Creazione e Stato Iniziale (2/2)

Verifiche iniziali:

- Controllare l'emissione dell'evento di creazione su **Etherscan**
- Verificare il valore di reputazione mediante la funzione *reputation()*
- Effettuare **l'incremento** della reputation con la funzione *reward()* → fatto tante volte finché non si supera 255 (massimo valore di reputation)



Dimostrazione di Overflow

Processo dettagliato:

1. Valore iniziale della reputazione: 200
2. Prima chiamata a reward(): 200 + 10 = 210
3. Continuo chiamate a reward() fino a 250 (con incremento di 10 per ogni chiamata)
4. Quinta chiamata a reward(): 250 + 10 = 4 (255 è il valore massimo per uint8 – ricomincia da 0)

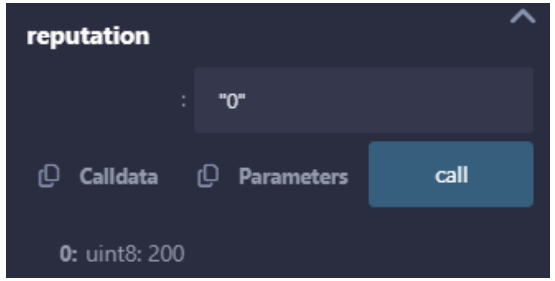
Spiegazione tecnica dell'overflow:

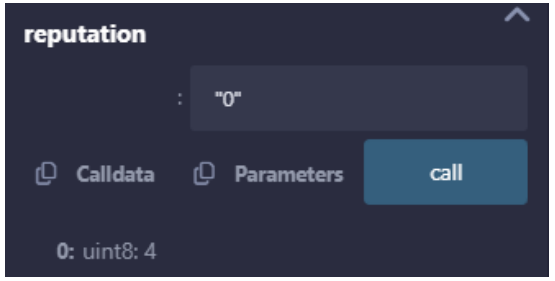
- In binario, 255 è rappresentato come 11111111 (8 bit tutti a 1)
- Aggiungere 10 (00001010) darebbe 100001001
- Poiché uint8 può memorizzare solo 8 bit, il bit più significativo viene troncato
- Risultato memorizzato: 00000100 (che è 4 in decimale)

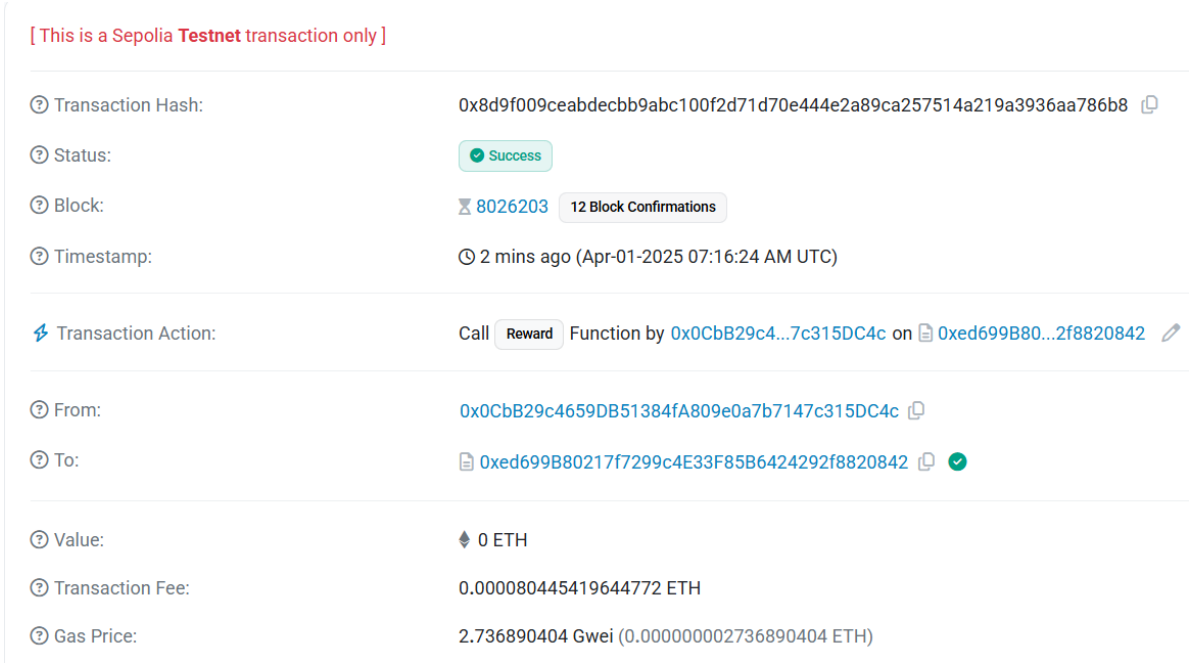
Dimostrazione di Overflow (2/2)

Impatto pratico:

- Un attaccante può artificialmente resettare una reputazione alta semplicemente chiamando reward() un numero sufficiente di volte
- Questo va a violare la logica di business, la quale prevede che le ricompense aumentino sempre la reputazione
- Come si può vedere, la reward è andata «a buon fine», facendo ripartire il valore di reputation da 0 (250 + 10 = 4).
    - Lo stesso discorso varrà per la funzione di penalize.



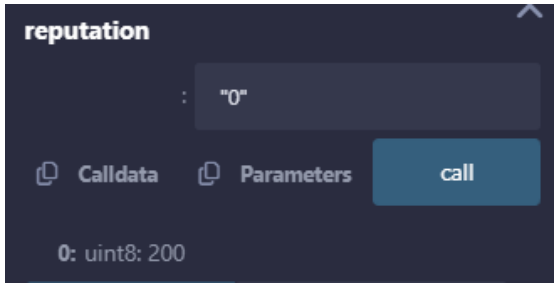


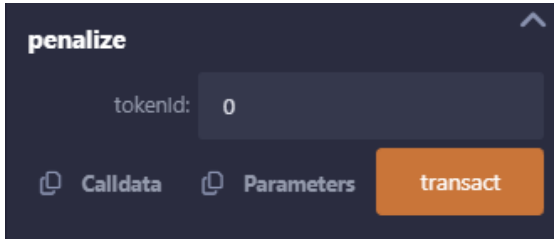


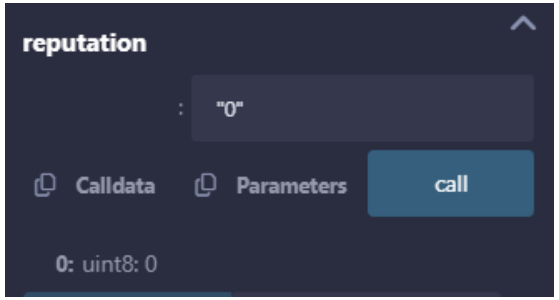
Dimostrazione di Underflow (1/2)

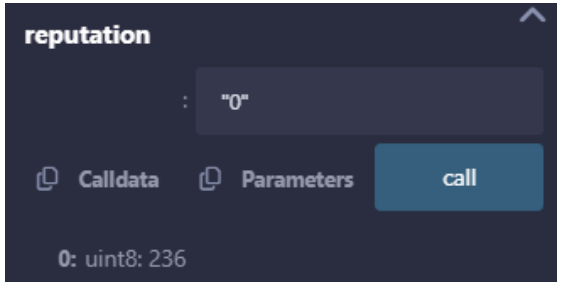
Processo dettagliato:

1. Creazione di un secondo NFT con reputazione iniziale 200
2. Prima chiamata a penalize(): 200 - 20 = 180
3. Continuare con le penalizzazioni fino a raggiungere il valore 0
4. Chiamata finale a penalize(): 0 - 20 = 236 (underflow!)









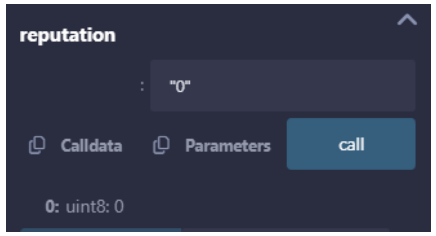
Spiegazione tecnica dell'underflow:

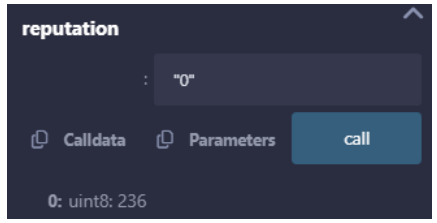
- Sottrarre 20 da 0 darebbe -20, che non è rappresentabile in un uint8
- In binario, 0 è 00000000
- Sottrarre 20 (00010100) richiederebbe un bit di segno per rappresentare il risultato negativo
- Poiché uint8 non ha bit di segno, il risultato "riparte" dall'alto: 11101100 (236 in decimale)

Dimostrazione di Underflow (2/2)

Impatto pratico:

- Un utente con reputazione bassa o zero può sfruttare questo bug per ottenere artificialmente una reputazione molto alta
- Questo sovverte completamente il sistema di penalizzazione previsto dal contratto





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

Transaction Hash:	0xa2b1179ad2165961bc3fc2b59e24d2addfcab52403d807e1c62fff1a5fcb4edf
Status:	Success
Block:	8069227 2 Block Confirmations
Timestamp:	28 secs ago (Apr-07-2025 09:32:12 AM UTC)
From:	0x0CbB29c4659DB51384fA809e0a7b7147c315DC4c
To:	0xAe040aE8a3C6726bbF66640ADc0197B284A28116
Value:	0 ETH
Transaction Fee:	0.002694445867778183 ETH
Gas Price:	99.975728833 Gwei (0.000000099975728833 ETH)

Impatto delle Vulnerabilità: conseguenze di sicurezza

1. **Manipolazione del sistema di reputazione**:
   - Gli utenti possono aggirare le penalizzazioni
   - Reputazioni elevate possono essere resettate o manipolate
   - Il sistema diventa inaffidabile come metrica di fiducia
2. **Impatti economici**:
   - Se il contratto associa privilegi o ricompense alla reputazione, questi possono essere indebitamente ottenuti
   - Potenziale perdita economica per il progetto e gli utenti onesti
3. **Compromissione della logica di business**:
   - Il contratto non funziona come previsto
   - Le decisioni basate sui valori di reputazione saranno errate
   - Perdita di fiducia nel sistema
4. **Caso reale di riferimento**:
   - Attacco BEC (Beauty Chain) dell'aprile 2018: a causa di un overflow, gli attaccanti hanno creato un numero enorme di token, portando la loro valutazione di mercato a zero

Mitigazione - Uso di require()

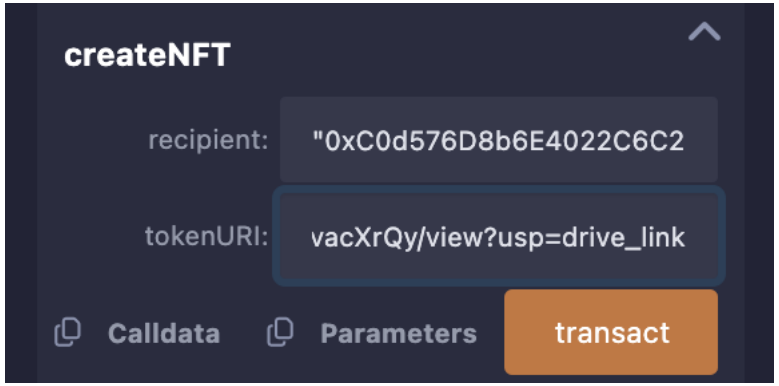
- Implementare **controlli preliminari** prima delle operazioni aritmetiche
- Utilizzare *require()* per verificare che le operazioni non causino overflow/underflow:
  - per *reward()* → si controlla che il valore di reputation sia al più 245 (poiché l'incremento è di 10);
  - per *penalize()* → si controlla che il valore di reputation sia almeno 20 (poiché il decremento è di 20)

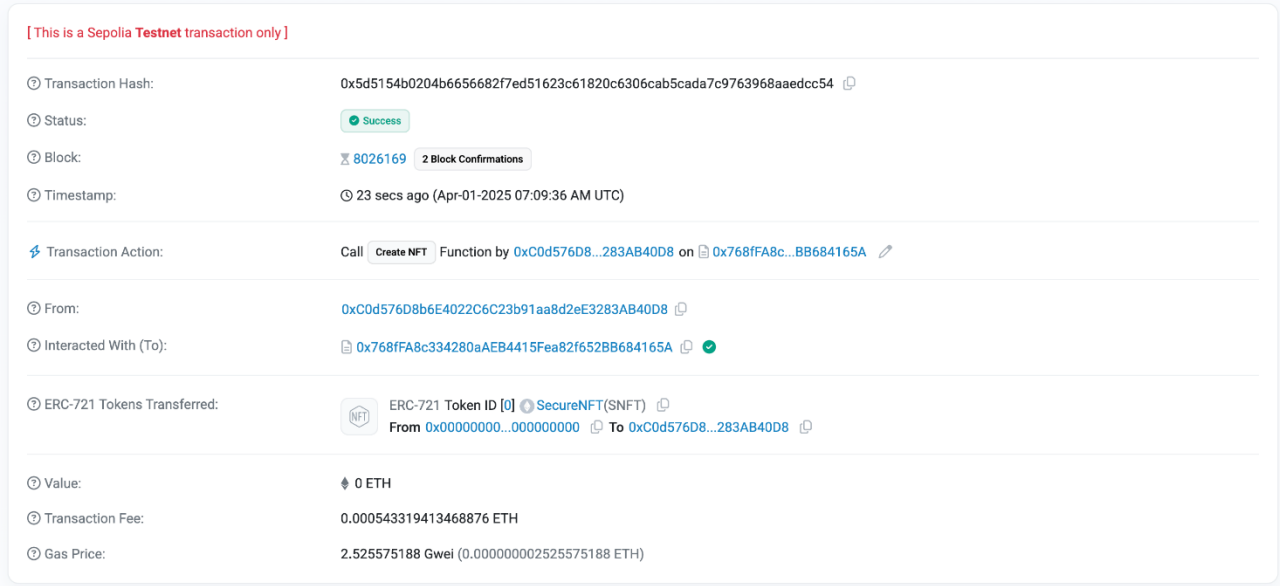
```
/// Reward con controllo su overflow
function reward(uint256 tokenId) public {    infinite gas
    require(ownerOf(tokenId) == msg.sender, "Not your NFT");
    require(reputation[tokenId] <= 245, "Overflow risk: max reputation reached");
    reputation[tokenId] += 10;
}

/// Penalize con controllo su underflow
function penalize(uint256 tokenId) public {    infinite gas
    require(reputation[tokenId] >= 20, "Underflow risk: too low to penalize");
    reputation[tokenId] -= 20;
}
```

Test della Mitigazione

- Esattamente come prima, viene prima di tutto creato un nuovo Dynamic NFT, utilizzando la classica funzione *createNFT():*



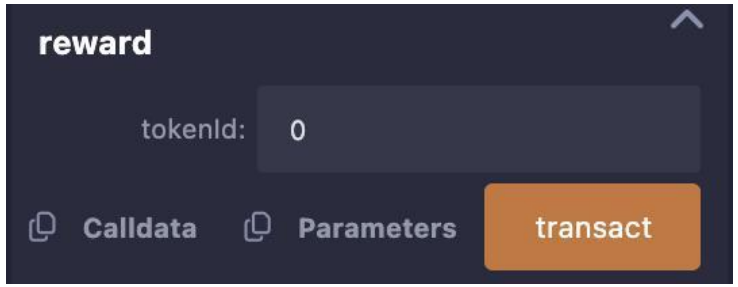


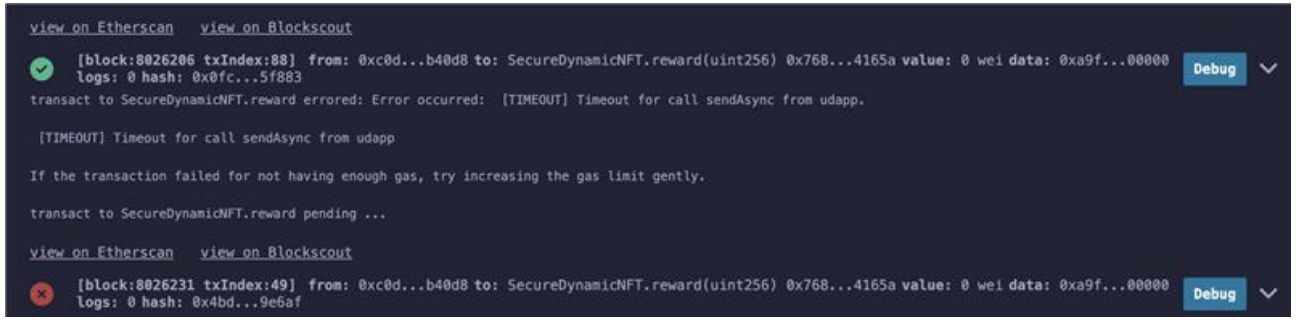
- Ciò è stato fatto per poter partire da un determinato **valore di reputation**, così da rendere più esplicito il meccanismo di mitigazione.

Test della Mitigazione – Overflow (1/2)

Verifica della protezione contro l'overflow:

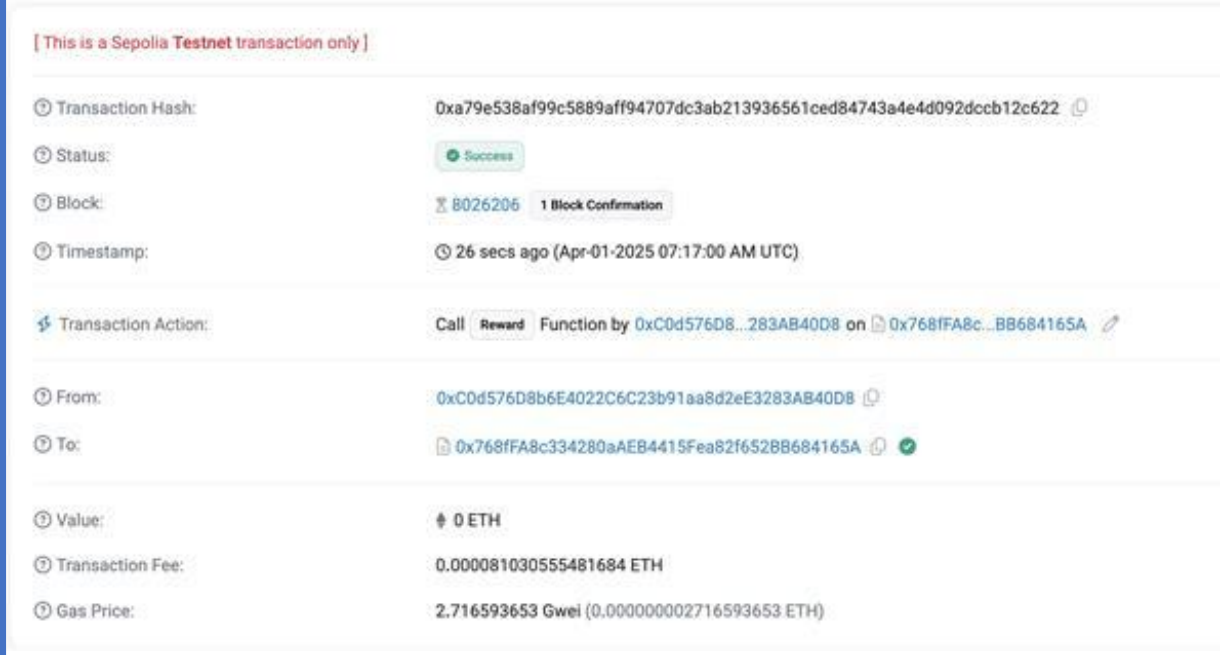
1. Partire da un NFT con reputazione elevata (235)
2. Chiamare *reward()* una prima volta: 235 + 10 = 245
3. Chiamo *reward()* tante volte fino a raggiungere il caso in cui dovrebbe verificarsi l'overflow (mitigato)
4. La transazione è *reverted* con il messaggio «**Overflow risk**» in quanto la vulnerabilità è mitigata
5. Si verifica che il valore della reputazione resta invariato



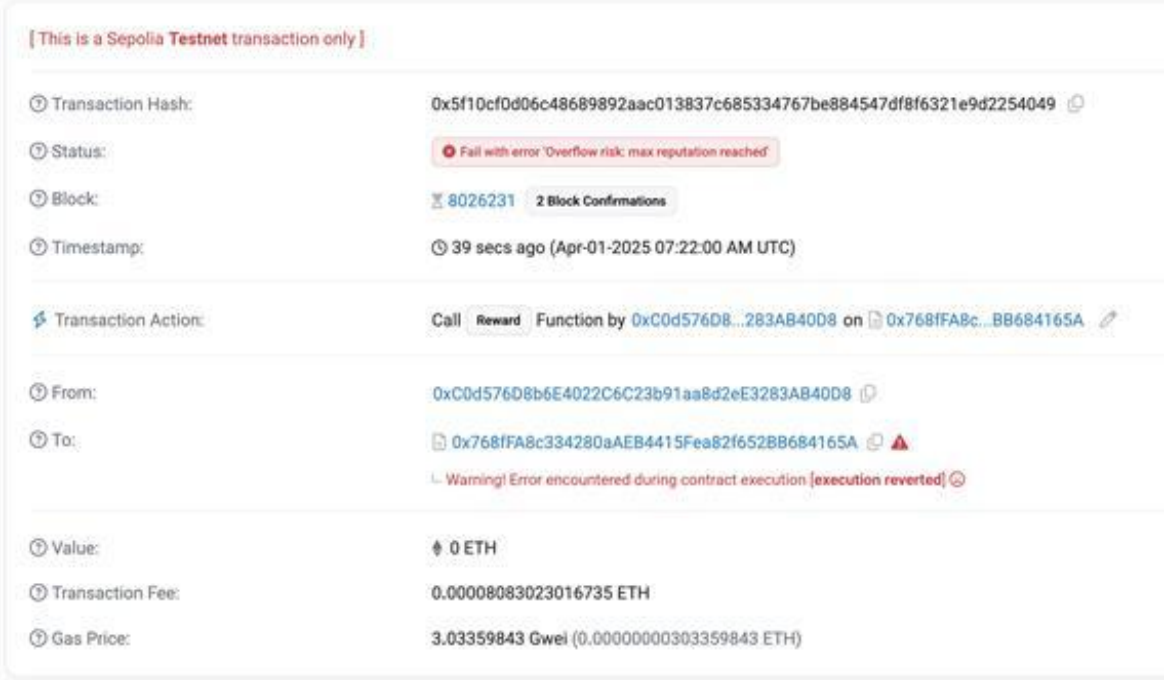


Test della Mitigazione – Overflow (2/2)

Prima volta



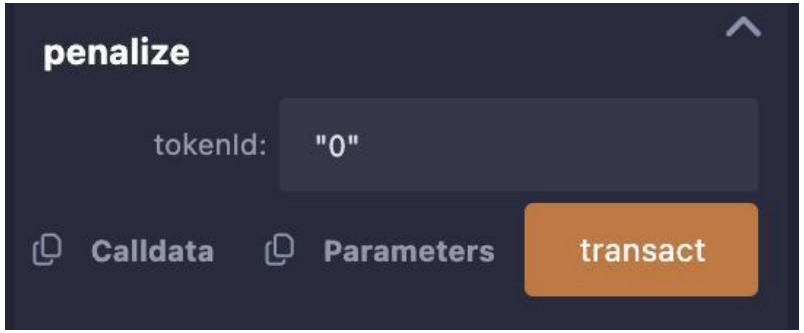
N-esima volta



Test della Mitigazione – Underflow (1/2)

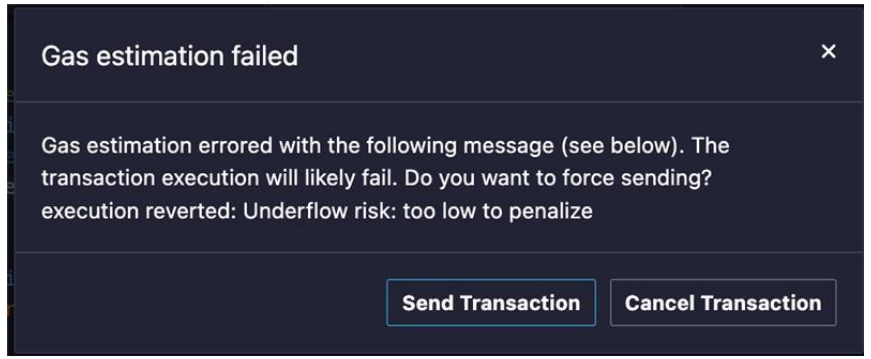
Verifica della protezione contro l'underflow:

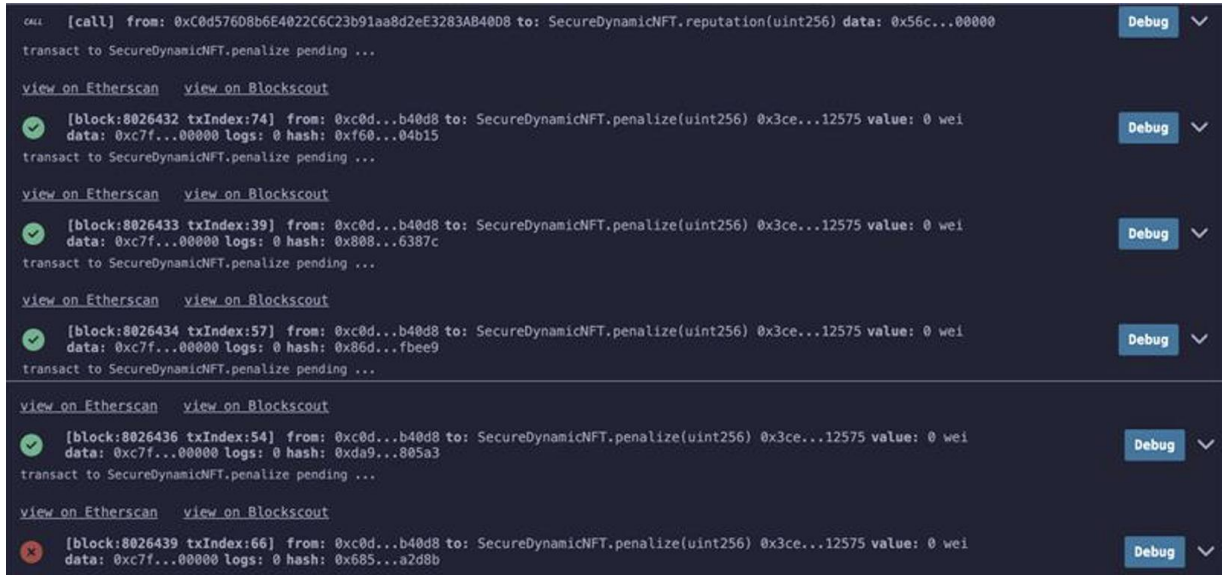
1. Partire da un NFT con reputazione bassa (es. 100)
2. Chiamare *penalize()*: 100 - 20 = 80
3. Chiamo penalize() tante volte fino a quella che causerebbe underflow
4. La transazione è *reverted* con il messaggio «**Underflow risk**» in quanto la vulnerabilità è mitigata
5. Si verifica che il valore della reputazione rimanga invariato

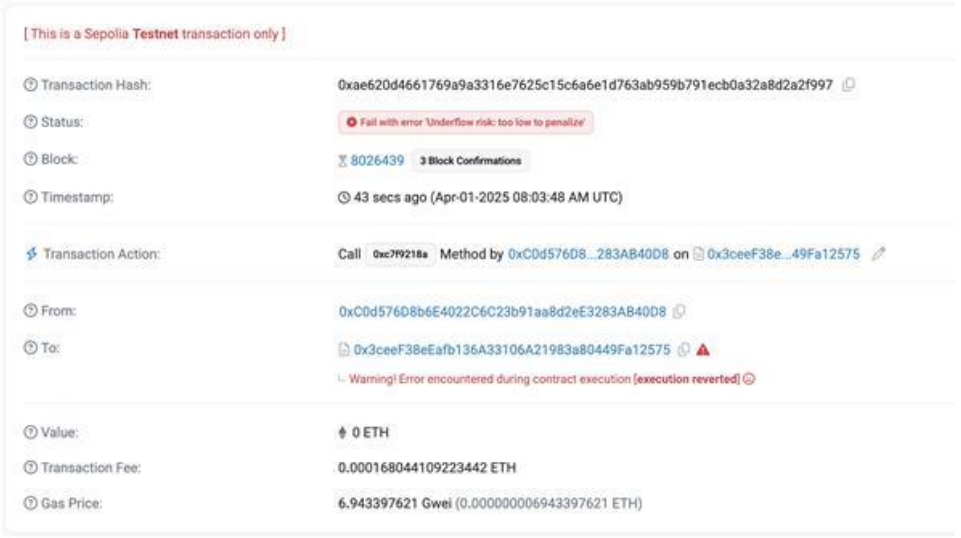


Test della Mitigazione – Underflow (2/2)

- La funzione *require()* **interrompe l'esecuzione** se la condizione non è soddisfatta
- Il gas viene consumato fino al punto di fallimento → qui l'attaccante andrà a «forzare la transazione» (**Send Transaction**)
- Nessuna modifica allo stato del contratto







Ulteriori Mitigazioni

1. Versioni moderne di Solidity

- **Solidity 0.8.0+**: Implementa controlli automatici per overflow/underflow
- Esempio: pragma solidity ^0.8.0; attiva questa protezione di default
- **Funzionamento**: Il compilatore inserisce automaticamente controlli simili ai nostri require()
- **Vantaggi**: Meno codice, minore rischio di errori, ottimizzazione del gas

2. Libreria SafeMath

- Per contratti che devono rimanere compatibili con versioni di Solidity < 0.8.0
- Implementa **funzioni matematiche sicure**, come add(), sub(), mul(), div()

3. Pattern di design sicuri

- Utilizzare **tipi di dati appropriati** (es. uint256 invece di uint8 se non necessario)
- Implementare limiti massimi e minimi espliciti per i valori

Conclusioni

Riepilogo delle lezioni apprese:

- Gli attacchi di overflow/underflow sono semplici da eseguire, ma potenzialmente devastanti
- I tipi di dati a dimensione fissa richiedono particolare attenzione nelle operazioni aritmetiche
- Le versioni moderne di Solidity (>=0.8.0) offrono protezione integrata

Best practice per lo sviluppo sicuro:

- Mantenere aggiornati compilatori e dipendenze
- Non disabilitare il controllo implicito delle ultime versioni di compilatori con unchecked (cosa fatta per mostrare la vulnerabilità)

N.B. Gli attacchi di overflow/underflow dimostrano l'importanza di comprendere i dettagli di basso livello quando si sviluppano smart contract.