Lorenzo Zaccagnini

Founder of Devoleum

Psychology MSc

Blockchain developer @Kaaja

Contatti
Lorenzo Zaccagnini Linkedin

Github
https://github.com/LorenzoZaccagnini

Ethereum

La seconda blockchain più usata dopo i Bitcoin, pensata per sviluppare applicazioni decentralizzate o DApp, la sua feature più importante è la possibilità di sviluppare Smart Contract.

- Possibilità di sviluppare applicazioni decentralizzate
- Scalabilità con l’implementazione del proof of stake
- Altamente documentata per lo sviluppo di applicazioni decentralizzate

Transazioni e contratti

Due tipi di account

- **Externally owned accounts** (controllati dalle persone)
- **Contract accounts** (controllati dagli smart contract)

Solidity

Partiamo dal colpevole

Solidity è un linguaggio di alto livello orientato agli oggetti per l'implementazione di smart contract. Gli smart contract sono programmi che regolano il comportamento degli account all'interno dello stato di Ethereum.

È influenzato da C++, Python e JavaScript ed è progettato per indirizzare la macchina virtuale Ethereum (EVM).

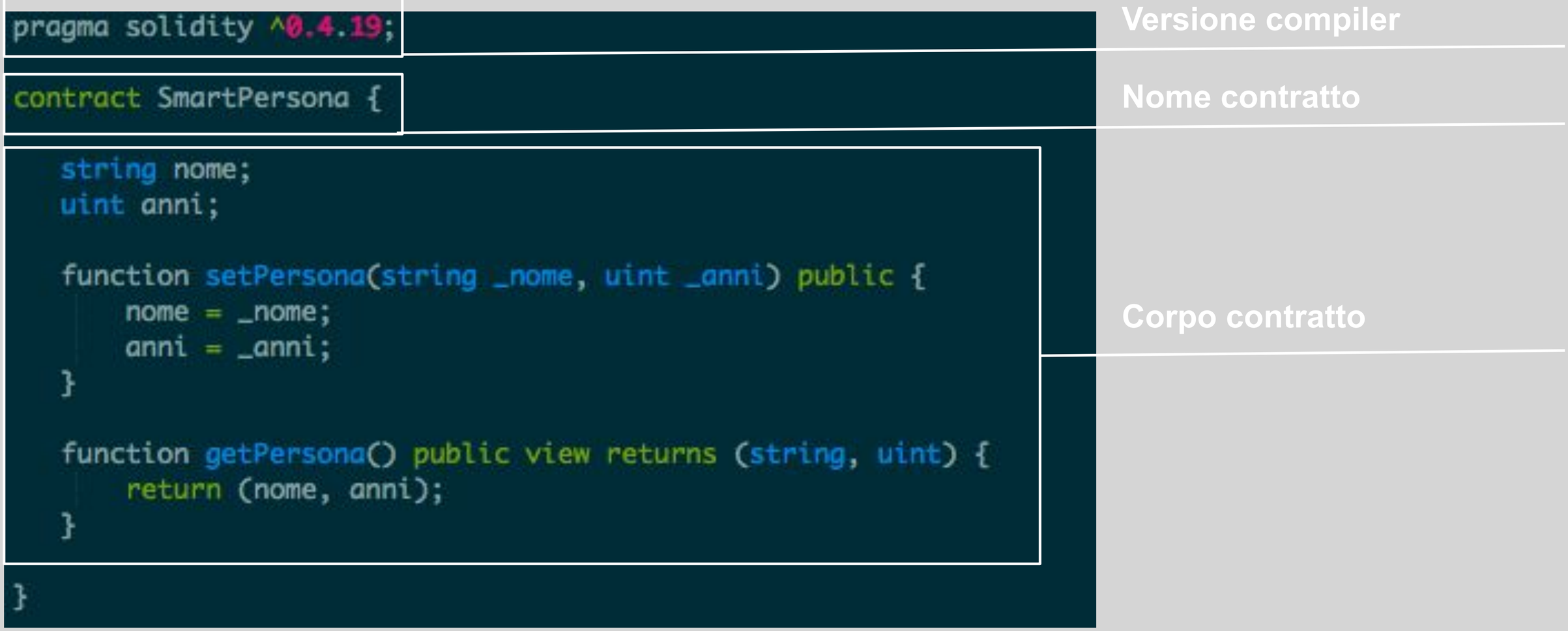
IMHO assomiglia moltissimo a C#

Esempio Smart Contract

Read and Store

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {
    uint256 number;
    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }
    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Anatomia di uno Smart Contract



Variabili globali principali

Ether Units: wei, finney, szabo, ether

Tempo: block.timestamp cioè adesso, seconds, minutes, hours, days, weeks, years

msg.sender: address di chi sta interagendo con lo smart contract

msg.value: quanta valuta è stata mandata

TxOrigin la variabile globale SWC-115

meglio usare msg.sender per usi più comuni

tx.origin è una variabile globale in Solidity che restituisce l'indirizzo del address che ha inviato la transazione. L'utilizzo della variabile per l'autorizzazione potrebbe rendere vulnerabile un contratto se un account autorizzato richiama un contratto malevolo.

```
pragma solidity 0.4.24;

contract MyContract {

    address owner;

    function MyContract() public {
        owner = msg.sender;
    }

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.transfer(amount);
    }

}
```

Visibilità delle funzioni

Tutti i dati nella blockchain sono pubblicamente visibili, possiamo solo decidere con quali funzioni dello smart contract si può interagire modificando la loro visibilità.

Public: accessibili internamente ed esternamente

Private: accessibili solo all’interno del contratto, esclusi i contratti derivati

Internal: accessibili all’interno dal contratto e dai suoi contratti derivati

External: accessibili solo esternamente, utili per ricevere grossi array

Overflow SWC-101

Calcoliamo questa somma

1 1 1 1 (15) +
0 0 0 1 (1)

Purtroppo non il risultato non è 16 ma:

0 0 0 0 (0)

Puoi pensare la matematica binaria come un orologio

Underflow SWC-101

Sottrazione

0 0 0 0 (0) -
0 0 0 1 (1)
1 1 1 1 (15)

Il risultato non è negativo ma il massimo della capacità che puo raggiungere, un unsigned integer può solo essere positivo!

Batch Overflow

L'exploit ha portato alla generazione di miliardi di token ERC20, causando l'interruzione temporanea dei depositi e dei prelievi di ERC20 da parte dei principali exchange fino a quando tutti i token non possono essere valutati per la vulnerabilità.

L'exploit, chiamato exploit batchOverflow, è stato osservato per la prima volta il 22 aprile 2018, quando sono stati creati 115 octodecilioni di BEC (Beauty Coin) in due transazioni. All'epoca, BEC veniva scambiato a $ 0,32 per token, il che rende il valore totale in dollari dell'exploit di circa $ 3,7 novemdecilioni, una cifra davvero assurda, che è un 1 seguito da 60 zeri.

Poloniex Exchange

@Poloniex

We've temporarily suspended ERC-20 token deposits and withdrawals while we review all smart contracts for exposure to the reported batchOverflow bug. We take any reports of vulnerabilities very seriously to ensure that customer funds remain safe. Thank you for your patience!

2:46 PM · Apr 25, 2018

419 68 Copy link to Tweet

Tweet your reply

Beauty Chain Exploit

https://www.youtube.com/watch?v=dK3ZXKVtYr4

Andiamo a vedere su remix

1. Check del balance
2. Oveflow con 57896044618658097711785492504343953926634992332820282019728792003956564819968
3. Il bilancio arriva a 0 bypassa i check e permette il mint indiscriminato

Prevenire attacchi su base matematica

Open zeppelin e le nuove versioni di solidity

- Usa SafeMath per prevenire overflow e underflow aritmetici
- Solidity 0.8 per impostazione predefinita genera un errore per overflow/underflow
- Usare modifier/require per restringere il range numerico

Quando fate affidamento ad altri contratti ricordate sempre di guardare la versione di pragma

Come si comprtano gli SC in ricezione?

Fallback e Receive

Prima di Solidity 0.6, la funzione di fallback era semplicemente una funzione anonima che assomigliava a questa:

function () external {}

Le versioni successive hanno diviso il comportamento in ricezione in due funzioni, fallback e receive

Receive

Un contratto può avere al massimo una funzione di ricezione, dichiarata utilizzando require() external payable{ ... } (senza la parola chiave function). Questa funzione non può avere parametri, non può restituire nulla e deve avere visibilità esterna e mutabilità dello stato payable.

La funzione **receive** viene eseguita su una chiamata al contratto con calldata vuoti. Questa è la funzione che viene eseguita su semplici trasferimenti Ether (ad esempio tramite .send() o .transfer()).

Se non esiste tale funzione, ma esiste una funzione di fallback pagabile, la funzione di fallback verrà chiamata su un semplice trasferimento Ether.

Se non è presente né una funzione di **receive** né una funzione di fallback pagabile, il contratto non può ricevere Ether tramite transazioni regolari e genera un'eccezione.

Fallback

Un contratto può avere al massimo una funzione fallback, dichiarata utilizzando f**allback() external [payable]** o **fallback (bytes calldata _input) external [payable] returns (bytes memory _output)** (entrambi senza la parola chiave function).

Questa funzione deve avere visibilità esterna. Una funzione di fallback può essere virtuale, può sovrascrivere e può avere modificatori.

La funzione di fallback viene eseguita su una chiamata al contratto se nessuna delle altre funzioni corrisponde alla firma della funzione chiamata, o se non sono stati forniti dati e non esiste una funzione Ether di ricezione. La funzione fallback riceve sempre i dati, ma per ricevere anche Ether deve essere contrassegnata come pagabile.

Se viene utilizzata la versione con parametri, _input conterrà i dati completi inviati al contratto (pari a msg.data) e potrà restituire i dati in _output. I dati restituiti non saranno codificati ABI. Verrà invece restituito senza modifiche (senza il padding).

Mandare ether forzatamente SWC-132

I contratti possono essere cancellati dalla blockchain chiamando selfdestruct.

Selfdestruct invia tutto l'Ether rimanente memorizzato nel contratto a un indirizzo designato. Un contratto malevolo può utilizzare selfdestruct per forzare l'invio di Ether a qualsiasi contratto.

Nel caso un contratto non sia protetto da questo tipo di evenienze si possono sballare bilanci e calcoli che riguardano il bilancio generale dello smart contract.

Gridlock

Il caso gridlock

https://medium.com/@nmcl/gridlock-a-smart-contract-bug-73b8310608a9

Delegation Attack

Spesso durante la scrittura dei contratti vorremmo chiamare funzioni all'interno di altri contratti per sfruttare le funzionalità all'interno dell'altro contratto o per motivi di aggiornabilità. Possiamo farlo sfruttando le librerie con le chiamate delegate. Ci sono vari motivi per farlo, incluso il risparmio sui costi di riutilizzo del codice evitando la ridistribuzione di librerie di grandi dimensioni.

Le chiamate delegate vengono utilizzate per chiamare la funzionalità del contratto, ma le modifiche si riflettono nel contesto del contratto chiamante. Si comporta essenzialmente come se si importasse la funzionalità del contratto logico nel contratto chiamante e le modifiche si riflettessero nel contesto del contratto chiamante. Si comporta in modo molto simile all'importazione di librerie quando si codificano progetti di grandi dimensioni e si utilizza tale funzionalità come se fosse parte del progetto.

Parity Hack

Analisi del codice

All'interno di parity wallet c'era una funzione di pagamento predefinita nota anche come funzione di fallback che utilizzava una chiamata delegata nella libreria del wallet. Le funzioni di fallback vengono chiamate quando viene effettuata una chiamata a un contratto e non viene specificata alcuna funzione durante l'invio di valore a un contratto. Utilizzando questa funzionalità, un utente è stato in grado di accedere alla funzione di fallback e sfruttare la chiamata delegata chiamando il contratto e non specificando una funzione ma specificando msg.data con il target e i valori mostrati nell'exploit precedente.

30 milioni di dollari andati via così

```
430     // gets called when no other function matches
431     function () payable {
432             // just being sent some cash?
433             if (msg.value > 0)
434                     Deposit (msg.sender, msg.value);
435             else if (msg.data.length > 0)
436                     _walletLibrary.delegatecall(msg.data);
```

Reentrancy SWC-107

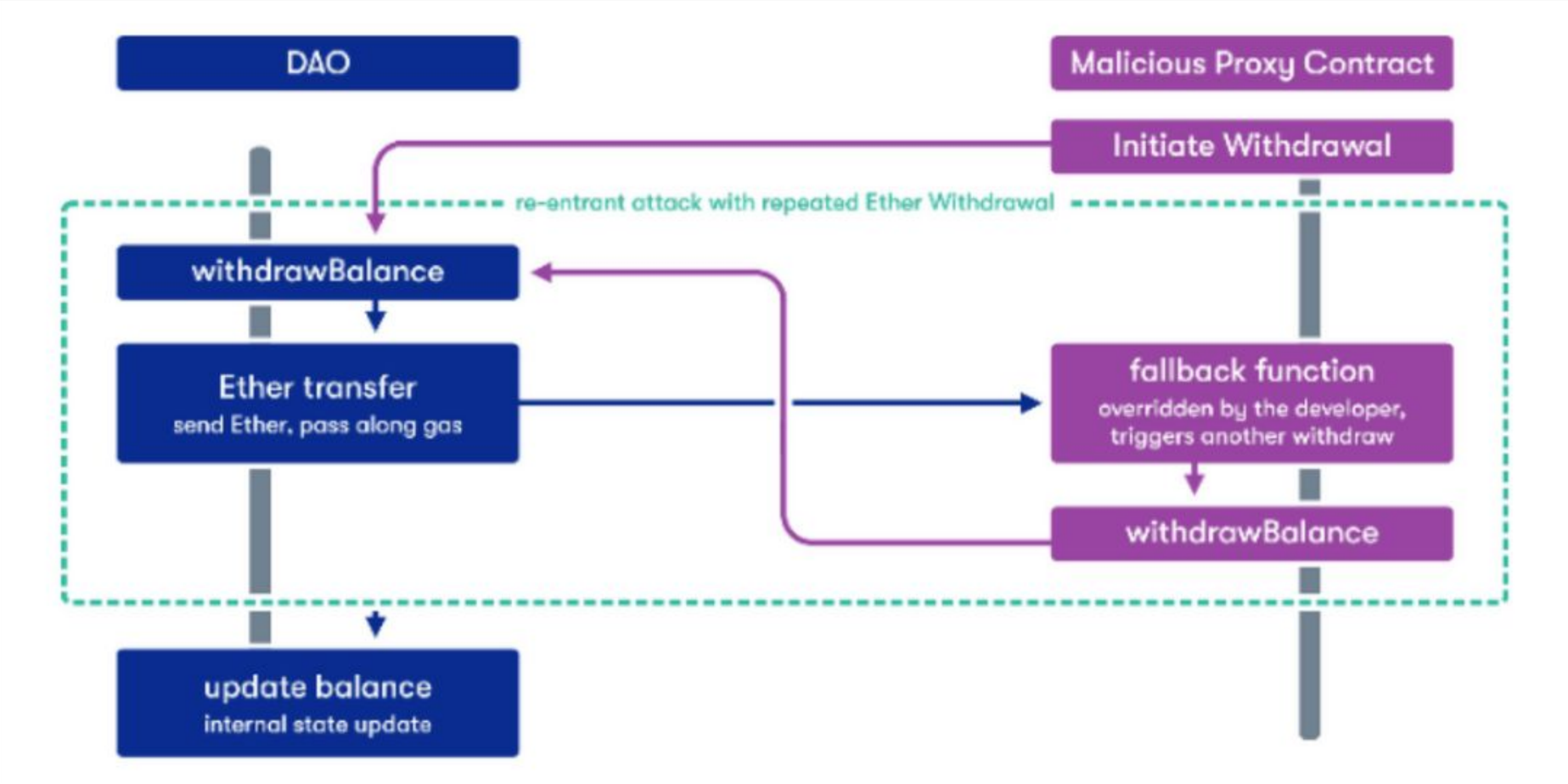
Scusa me lo puoi ripetere?

Una procedura è rientrante se la sua esecuzione può essere interrotta nel mezzo, avviata (reinserita) ed entrambe le esecuzioni possono essere completate senza errori di esecuzione. Nel contesto dei contratti di Ethereum, la reentrancy può portare a gravi vulnerabilità.

L'esempio più famoso di questo è stato il DAO Hack, in cui sono stati sottratti 70 milioni di dollari di Ether.

Dao Hack

Analisi codice



Dao Hack

Analisi codice

```
1  contract BasicDAO {
2
3    mapping (address => uint) public balances;
4    ...
5
6    // transfer the entire balance of the caller of this function
7      // to the caller
8    function withdrawBalance() public {
9      bool result = msg.sender.call.value(balances[msg.sender])();
10     if (!result) {
11       throw;
12     }
13   // update balance of the withdrawer.
14     balances[msg.sender] = 0;
15   }
16 }
```

Checks-Effects-Interactions Pattern

La maggior parte delle funzioni eseguirà prima alcuni controlli (chi ha chiamato la funzione, ha inviato abbastanza Ether, la persona ha token, ecc.). Questi controlli dovrebbero essere fatti prima.

Come secondo passaggio, se tutti i controlli sono stati superati, dovrebbero essere apportati effetti alle variabili di stato del contratto in corso. L'interazione con altri contratti dovrebbe essere l'ultimo passo in qualsiasi funzione.

Le chiamate verso contratti noti potrebbero a loro volta causare chiamate a contratti sconosciuti, quindi probabilmente è meglio applicare sempre questo schema.

Visibilità e interazioni

La visibilità è sempre pubblica ma cambiano le interazioni

Solidity conosce due tipi di chiamate di funzione: quelle interne che non creano una vera chiamata EVM (chiamata anche "chiamata di messaggio") e quelle esterne che lo fanno. Per questo motivo, esistono quattro tipi di visibilità per le funzioni e le variabili di stato. Le funzioni devono essere specificate come esterne, pubbliche, interne o private. Per le variabili di stato, esterno non è possibile.

esterna

Le funzioni esterne fanno parte dell'interfaccia del contratto, il che significa che possono essere chiamate da altri contratti e tramite transazioni. Una funzione esterna f non può essere chiamata internamente (cioè f() non funziona, ma this.f() funziona).

pubblica

Le funzioni pubbliche fanno parte dell'interfaccia del contratto e possono essere chiamate internamente o tramite messaggi. Per le variabili di stato pubbliche, viene generata una funzione getter automatica.

interna

Tali funzioni e variabili di stato sono accessibili solo internamente (cioè dall'interno del contratto in corso o dei contratti che ne derivano), senza utilizzarlo. Questo è il livello di visibilità predefinito per le variabili di stato.

privata

Le funzioni private e le variabili di stato sono visibili solo per il contratto in cui sono definite e non nei contratti derivati.

https://docs.soliditylang.org/en/latest/contracts.html#visibility-and-getters

Variabili private

Ma sono davvero private?

Parity hack

Visibilità

Un utente ha sfruttato la visibilità pubblica di una funzione del wallet che doveva essere interna, purtroppo questa funzione era l'assegnazione della ownership

https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede4bb083def75ec

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104  104
105  105    // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106  106    // as well as the selection of addresses capable of confirming them.
107       -  function initMultiowned(address[] _owners, uint _required) {
     107  +  function initMultiowned(address[] _owners, uint _required) internal {
108  108      m_numOwners = _owners.length + 1;
109  109      m_owners[1] = uint(msg.sender);
110  110      m_ownerIndex[uint(msg.sender)] = 1;
```

Entropia

La forza del caos è la nostra sicurezza

Nella crittografia e nella cybersecurity in generale la casualità è un fattore importante per la sicurezza, purtroppo su Ethereum non abbiamo la possibilità di avere numeri casuali nativamente.

Quando le transazioni vengono minate, devono essere confermate da più di un nodo sulla rete. Ciò significa che ogni nodo deve giungere alla stessa conclusione. Quindi, se una funzione fosse veramente casuale, ogni nodo giungerebbe a una conclusione diversa, risultando in una transazione non confermata

Succede non casualmente

SWC-120

Arseny Reutov ha scritto un post dopo aver analizzato 3649 contratti che utilizzavano una sorta di generatore di numeri pseudo casuali (PRNG) e ha trovato 43 smart contract vulnerabili.

https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620

Interoperatività - Oracoli

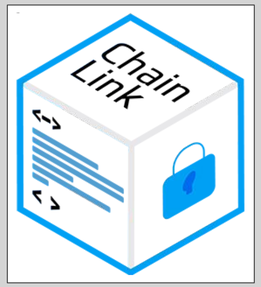
Il problema

Gli smart contract di base non possono connettersi all’esterno ma per funzionare hanno bisogno di dati, molte volte questi dati risiedono all’esterno della blockchain, come può capitare per lo stato di una spedizione oppure un tasso di cambio.

La soluzione

Gli oracoli sono degli agenti che trovano e verificano eventi all’esterno oppure all’interno della blockchain, permettendo agli smart contract di agire ed interagire con il mondo all’esterno della blockchain.

Oracoli - Scenario attuale



Chain link https://chain.link/

Chain link ci permette di gestire in maniera sicura dati all’esterno della blockchain, utilizzando tecnologie come Town Crier, la quale ci permette di eseguire le nostre operazioni tra ethereum e mondo esterno in un ambiente isolato, confidenziale, essendo sicuri dell’autenticità del nostro interlocutore.

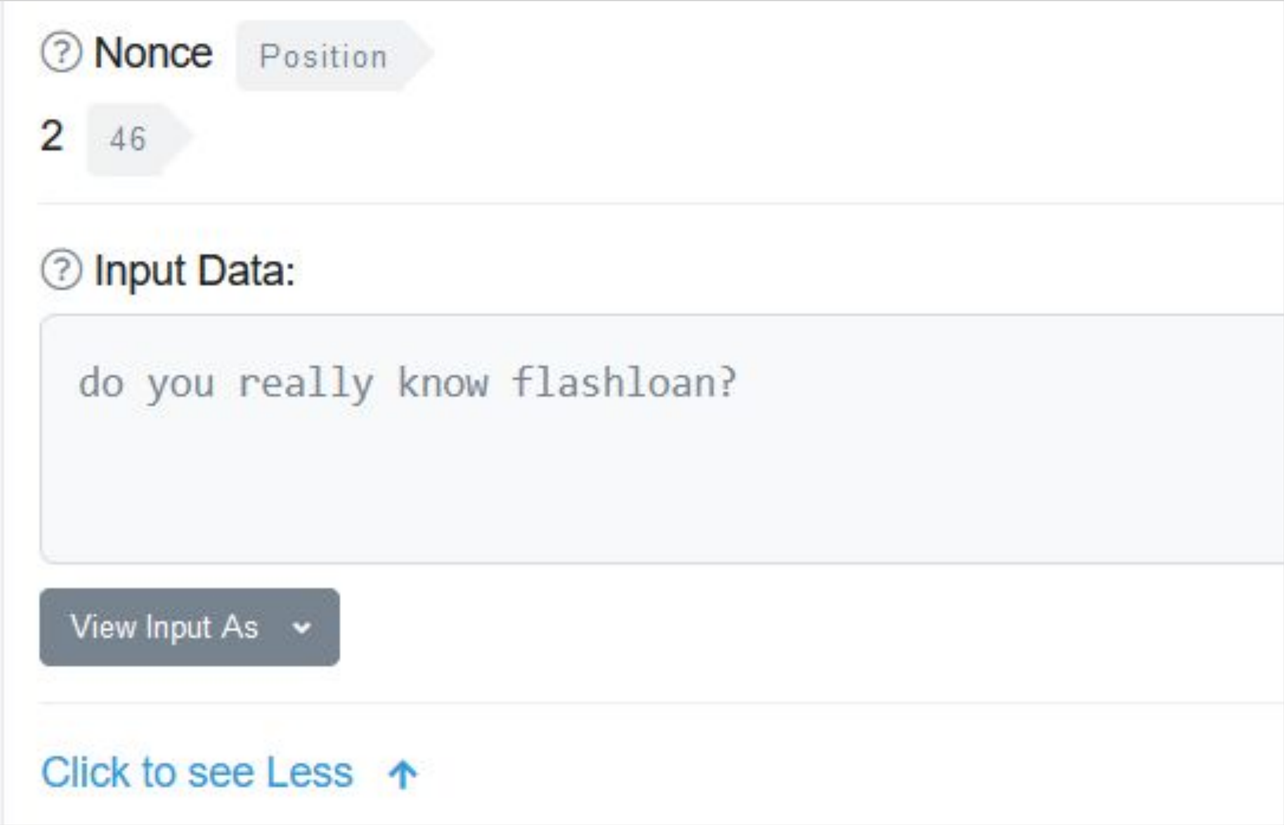
Grazie a questo servizio è possibile far reagire gli smart contract a:

- Spedizioni
- Voli
- Pagamenti fiat
- Fluttuazioni di prezzi

Flash loan

Un prestito che si può rifiutare

Il flashloan è un prestito che si risolve in un’unica transazione, se non viene ripagato entro essa la transazione fallisce ed i soldi tornano tutti indietro.



Flashloan attack

La mossa della balena

I flashloan permettono tramite il concetto di “arbitraggio” di guadagnare da differenze di prezzo degli stessi asset su exchange decentralizzati diversi.

Abusando di questo concetto è possibile portare a termine un attacco flashloan, cioè prendere in prestito un’enorme quantità di token e sfruttare il capitale gigantesco per sbilanciare il mercato in maniera prevedible e sbancare tutto sfruttando l’”arbitraggio”.

Uno dei più famosi fu quello Pancake Swap.

https://medium.com/amber-group/bsc-flash-loan-attack-pancakebunny-3361b6d814fd

Agave - Hundred

Rientranza

Gli attacchi sono stati resi possibili grazie alla progettazione del token xDAI che contiene la funzione callAfterTransfer() creando una vulnerabilità di rientro.

Utilizzando flash loan come garanzia iniziale, gli aggressori hanno annidato funzioni di prestito aggiuntive l'una nell'altra, aumentando l'importo preso in prestito prima che il protocollo potesse aggiornare il saldo del debito. La ripetizione di questo processo ha portato a prendere in prestito attività di valore molto più elevato della garanzia fornita.

Il vettore di attacco è lo stesso del caso da 18,8 milioni di dollari di CREAM Finance lo scorso agosto.

Dove esercitarsi?

Ethernaut e DamnDefi

https://www.damnvulnerabledefi.xyz/

https://ethernaut.openzeppelin.com/

Come testare gli smart contract?

Static analysis

Possiamo usare tool come smartcheck, slither, maian.

La cosa più comoda è creare una pipeline di test sia unit test (mocha chai) e analisi statiche fatte da più tool, magari integrandola ad ogni aggiornamento del codice del contratto.

Referenze utili:
https://swcregistry.io/docs/SWC-107
https://consensys.net/diligence/tools/