



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

Descrizione SC. Solodity

———Quale vulnerabilità è presente in questo smart contract in Solidity?
Descrivere una possibile soluzione:

```
pragma solidity ^0.4.18;

contract Token {
    mapping( address => uint ) balances;

    uint public totalSupply;

    function Token( uint _initialSupply ) {
        balances[msg.sender] = totalSupply = _initialSupply;
    }

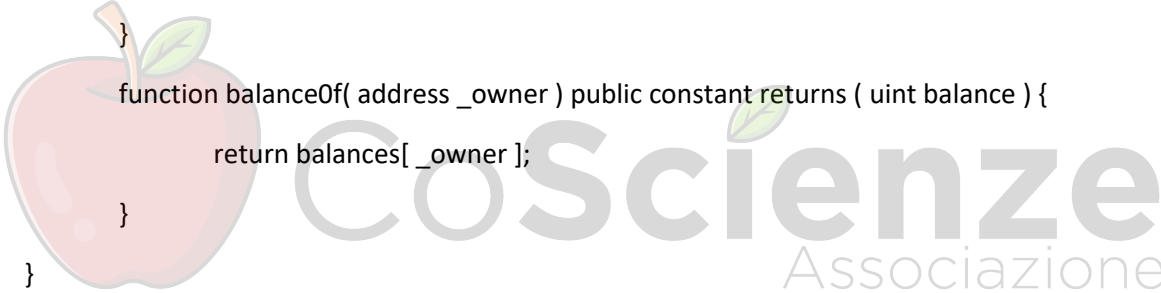
    function transfer( address _to, uint _value ) public returns ( bool ) {
        riga [9]      require( balances[ msg.sender ] - value >= 0 );

        balances[msg.sender] -= _value;

        balances[ _to ] += _value;

        return true;
    }

    function balanceOf( address _owner ) public constant returns ( uint balance ) {
        return balances[ _owner ];
    }
}
```



RISPOSTA:

Questo contratto è un token che consente agli utenti di trasferire Ether mediante la funzione transfer().

ERRORE:

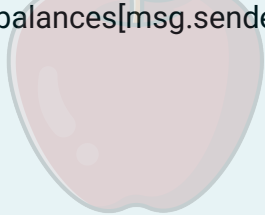
Si consideri un utente che non ha alcun saldo e invoca la funzione transfer() con qualsiasi _value diverso da zero. L'istruzione require alla riga [9] viene superata perché balances[msg.sender] è zero (e un uint256) quindi la sottrazione con un qualsiasi importo positivo (escluso 2^{256}) risulterà in un numero positivo a causa dell'underflow. Ciò vale anche per la riga [9], dove al saldo verrà accreditato un numero positivo.

L'istruzione require nella riga [9] può essere bypassata utilizzando un underflow.

La soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare o costruire librerie matematiche sicure come SafeMath di Open Zeppelin.

Quale vulnerabilità è presente in questo smart contract in Solidity? Descrivere una possibile soluzione:

```
contract TimeLock {  
  
    mapping(address => uint) public balances;  
    mapping(address => uint) public lockTime;  
  
    function deposit() public payable {  
        balances[msg.sender] += msg.value;  
        lockTime[msg.sender] = now + 1 weeks;  
    }  
  
    function increaseLockTime(uint _secondsToIncrease) public {  
        lockTime[msg.sender] += _secondsToIncrease;  
    }  
  
    function withdraw() public {  
        require(balances[msg.sender] > 0);  
        require(now > lockTime[msg.sender]);  
        msg.sender.transfer(balances[msg.sender]);  
        balances[msg.sender] = 0;  
    }  
}
```



CoScienze
Associazione

Sostanzialmente questo smart contract rappresenta un portafoglio a tempo.

Essenzialmente una volta depositata una cifra, essa non si potrà prelevare finché non si verifica l'attesa impostata come lockTime, tuttavia lockTime è rappresentata con un uint.

Questo è una vulnerabilità in quanto la virtual Machine di ethereum ha un intervallo di rappresentazione che va da [0,255].

Praticamente nel caso in cui una variabile uint8 abbia valore 255, nel caso sommassimo 1, essa avrà valore 0, questa situazione è detta Overflow.

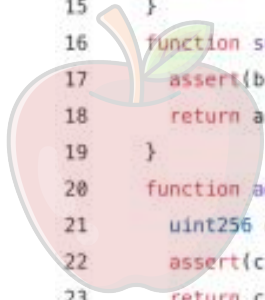
Praticamente un malintenzionato, sfruttando il fatto che lockTime è dichiarata pubblica, può aprire il valore. Tramite un contratto esterno potrebbe dunque chiamare increaselockTime ed azzerare l'attesa inerente al prelievo, semplicemente utilizzando una variabile da sommare a lock time, così che si azzeri e si possa prelevare Ether immediatamente.

Per risolvere questa problematica bisogna utilizzare delle apposite librerie matematiche, che non permettono di poter sfruttare questa vulnerabilità, in particolare le operazioni di add o di

sottrazione ed altro, devono essere utilizzate per mezzo di funzioni di libreria, come ad esempio SafeMath di Open Zeppelin.

```
1  library SafeMath {
2      function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3          if (a == 0) {
4              return 0;
5          }
6          uint256 c = a * b;
7          assert(c / a == b);
8          return c;
9      }
10     function div(uint256 a, uint256 b) internal pure returns (uint256) {
11         // assert(b > 0); // Solidity automatically throws when dividing by 0
12         uint256 c = a / b;
13         // assert(a == b * c + a % b); // There is no case in which this doesn't hold
14         return c;
15     }
16     function sub(uint256 a, uint256 b) internal pure returns (uint256) {
17         assert(b <= a);
18         return a - b;
19     }
20     function add(uint256 a, uint256 b) internal pure returns (uint256) {
21         uint256 c = a + b;
22         assert(c >= a);
23         return c;
24     }
25 }
```

La soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare o costruire librerie matematiche sicure come SafeMath di Open Zeppelin.



CosScienze
Associazione

HASHFORETHER

```
1  contract HashForEther {
2
3      function withdrawWinnings() {
4          // Winner if the last 8 hex characters of the address are 0.
5          require(uint32(msg.sender) == 0);
6          _sendWinnings();
7      }
8
9      function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

deve generare un i
ultimi 8 caratteri esad

Questo semplice contratto è progettato per fungere da gioco per indovinare gli indirizzi. Per vincere il saldo del contratto, un utente deve generare un indirizzo Ethereum i cui ultimi 8 caratteri esadecimali sono 0.

Purtroppo non è stata specificata la visibilità delle funzioni, e chiunque può invocare la funzione `_sendWinnings()`, che è pubblica, e ottenere la taglia non rispettando il gioco.

SOLUZIONE:

E' buona norma specificare sempre la visibilità di tutte le funzioni in un contratto, anche se intenzionalmente pubbliche.

LOTTO

```
1  contract Lotto {
2      bool public payedOut = false;
3      address public winner;
4      uint public winAmount;
5
6      // ... extra functionality here
7      function sendToWinner() public {
8          require(!payedOut);
9          winner.send(winAmount);
10         payedOut = true;
11     }
12
13     function withdrawLeftOver() public {
14         require(payedOut);
15         msg.sender.send(this.balance);
16     }
17 }
```

Qua

ERRORE:

Viene utilizzata una `send()` senza controllarne la risposta. Un vincitore la cui transazione fallisce (o esaurendo il gas, o perché invoca `throws` intenzionalmente nella fallback o tramite un `call stack depth attack`, impossibile su recenti EVM) consente a `payedOut` di essere impostato su `true` (indipendentemente dal fatto che ether sia stato inviato o no). In questo caso, chiunque può ritirare la vincita tramite la funzione `drawLeftOver()`.

SOLUZIONE:

Quando possibile, si deve preferire `transfer()` invece di `send()`, poiché quest'ultima propaga chi errori all'interno verso il chiamante disfacendo la transazione. Se `send()` è richiesto, assicurati sempre di controllare il valore restituito, dove è importante che l'errore venga gestito nel contratto senza annullare tutte le modifiche di stato.

FINDTHISHASH

```
1  contract FindThisHash {
2      bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44
3
4      constructor() public payable {} // load with ether
5
6      function solve(string solution) public {
7          // If you can find the pre image of the hash, receive 1000 ether
8          require(hash == sha3(solution));
9          msg.sender.transfer(1000 ether);
10     }
11 }
```

ERRORE

L'utente che trova la stringa per l'hash sha3 nella variabile `hash` può inviare la soluzione e recuperare il 1000 Ether. Ipotizziamo che un utente capisce che la soluzione è `Ethereum!`, e invoca di conseguenza `solve()`. Un utente malintenzionato può aver controllato il pool di transazioni e vedono questa soluzione,

invia una transazione equivalente con un prezzo del gas molto più alto rispetto alla quella originale. A causa del gas più elevato, la seconda transazione sarà accetterà prima - `front-running attack`. L'attaccante otterrà i 1000 Ether e l'utente onesto non otterrà nulla.

SOLUZIONE:

Tra le possibili soluzioni, uno robusto è di utilizzare uno schema di `commit-reveal`, quando possibile.

- Gli utenti inviano transazioni con informazioni nascoste (in genere un hash).
- Dopo che la transazione è stata inclusa in un blocco, l'utente invia un'altra transazione rivelando i dati che sono stati inviati.

Questo metodo impedisce sia ai miner che agli utenti di effettuare transazioni anticipate poiché non possono determinare il contenuto della transazione.

DISTRIBUTETOKENS

```
1  contract DistributeTokens {
2      address public owner; // gets set somewhere
3      address[] investors; // array of investors
4      uint[] investorTokens; // the amount of tokens each investor gets
5
6      // ... extra functionality, including transfertoken()
7
8      function invest() public payable {
9          investors.push(msg.sender);
10         investorTokens.push(msg.value * 5); // 5 times the wei sent
11     }
12
13     function distribute() public {
14         require(msg.sender == owner); // only owner
15         for(uint i = 0; i < investors.length; i++) {
16             // here transferToken(to,amount) transfers "amount" of tokens to the address
17             transferToken(investors[i], investorTokens[i]);
18         }
19     }
20 }
```

Quale errore è p

ERRORE:

Si noti che il ciclo in questo contratto viene eseguito su un array che può essere modificato arbitrariamente. Un utente malintenzionato può creare molti account in modo tale che il gas richiesto superi il limite del gas di blocco, rendendo sostanzialmente inutilizzabile la funzione distribute().

SOLUZIONE:

La soluzione consiste nel evitare di ciclare su strutture manipolate esternamente, oppure nell'applicare il withdrawal pattern: ogni utente deve invocare una funzione isolata che richiede il trasferimento del token.

ROULETTE

```
1 contract Roulette {
2     uint public pastBlockTime; // Forces one bet per block
3
4     constructor() public payable {} // Initially fund contract
5
6     // fallback function used to make a bet
7     function () public payable {
8         require(msg.value == 10 ether); // must send 10 ether to play
9         require(now != pastBlockTime); // only 1 transaction per block
10        pastBlockTime = now;
11        if(now % 15 == 0) { // winner
12            msg.sender.transfer(this.balance);
13        }
14    }
15 }
```

Ques
semp
può :

Questo contratto si comporta come una semplice lotteria. Una transazione per blocco può scommettere 10 ether per avere la possibilità di vincere il saldo del contratto. Il presupposto è che `block.timestamp` sia distribuito uniformemente sulle ultime due cifre. Se così fosse, ci sarebbe una probabilità di 1/15 di vincere questa lotteria.

Il margine d'azione è però limitato: i timestamp aumentano in modo monotono e non si possono scegliere valori arbitrari o non troppo lontano nel futuro poiché il blocco risulterebbe rifiutato dalla rete.

Se nel contratto è presente una quantità sufficiente di Ether, un minatore che risolve un blocco è incentivato a scegliere un timestamp tale che `block.timestamp` o `now` modulo 15 sia 0. In tal modo può vincere il saldo del contratto insieme alla ricompensa per il mining del blocco. Poiché c'è solo un partecipante alla lotteria per blocco, si verifica il caso del front-running attack.

OWNERWALLET

```
1 contract OwnerWallet {
2     address public owner;
3     //constructor
4     function ownerWallet(address _owner) public {
5         owner = _owner;
6     }
7
8     // fallback. Collect ether.
9     function () payable {}
10
11    function withdraw() public {
12        require(msg.sender == owner);
13        msg.sender.transfer(this.balance);
14    }
15 }
```

Quale errore è

ERRORE:

Il nome del costruttore non corrisponde con quello del contratto, e quest'ultimo può essere invocato da

qualunque per configurarsi come owner.

SOLUZIONE:

Questo problema è stato risolto nel compilatore Solidity nella versione 0.4.22, introducendo una parola chiave constructor che specifica il costruttore, invece di richiedere che il nome della funzione corrisponda al nome del contratto.

NAMEREGISTRAR

```
1 // A Locked Name Registrar
2 contract NameRegistrar {
3     bool public unlocked = false; // registrar locked, no name updates
4
5     struct NameRecord { // map hashes to addresses
6         bytes32 name;
7         address mappedAddress;
8     }
9     mapping(address => NameRecord) public registeredNameRecord; // records who reg
10    mapping(bytes32 => address) public resolve; // resolves hashes to addresses
11
12    function register(bytes32 _name, address _mappedAddress) public {
13        // set up the new NameRecord
14        NameRecord newRecord;
15        newRecord.name = _name;
16        newRecord.mappedAddress = _mappedAddress;
17        resolve[_name] = _mappedAddress;
18        registeredNameRecord[msg.sender] = newRecord;
19        require(unlocked); // only allow registrations if contract is unlocked
20    }
21 }
```


Quando il contratto è sbloccato, consente a chiunque di registrare un nome (come hash bytes32) e mappare quel nome a un indirizzo. Sfortunatamente, è inizialmente bloccato e la richiesta alla linea [19] impedisce a register() di aggiungere record di nomi.

I tipi struct definiscono riferimenti e newRecord non è inizializzato con new.

Solidity inizializza per impostazione predefinita istanze di tipi di dati complessi, come gli struct, a storage quando vengono usati come variabili locali. newRecord punta a storage e newRecord.name a slot[0], ovvero ad unlocked, modificabile direttamente mediante il parametro _name della funzione register().

Il compilatore Solidity genera dei warning quando trova delle variabili non inizializzate, quindi gli sviluppatori dovrebbero prestare molta attenzione a questi avvisi. L'attuale versione di mist (0.10) non consente la compilazione di questi contratti.

PISHABLE ATTACKCONTRACT



```
1 contract Phishable {
2     address public owner;
3
4     constructor (address _owner) {
5         owner = _owner;
6     }
7
8     function () public payable {} // collect ether
9     function withdrawAll(address _recipient) public {
10         require(tx.origin == owner);
11         _recipient.transfer(this.balance);
12     }
13 }

1 import "Phishable.sol";
2 contract AttackContract {
3
4     Phishable phishableContract;
5     address attacker; // The attackers address to receive funds.
6     constructor (Phishable _phishableContract, address _attackerAddress) {
7         phishableContract = _phishableContract;
8         attacker = _attackerAddress;
9     }
10
11     function () {
12         phishableContract.withdrawAll(attacker);
13     }
14 }
```

Solidity ha una variabile globale, `tx.origin` che attraversa l'intero stack di chiamate e restituisce l'indirizzo dell'account che ha originariamente inviato la chiamata (o la transazione). L'utilizzo di questa variabile per l'autenticazione negli smart contract lascia il contratto vulnerabile a un attacco phishing.

Un attaccante può convincere il proprietario del contratto Phishable di inviargli dell'Ether su un indirizzo, senza riconoscere che corrisponde a AttackContract. L'effetto è di chiamare la funzione di fallback, che invoca quella del contratto vittima.

`tx.origin` non deve essere utilizzato per l'autorizzazione negli smart contract, ma per altri scopi. Ad esempio, se si volesse negare ai contratti esterni di chiamare il contratto corrente, si potrebbe implementare una `require (tx.origin == msg.sender)`.