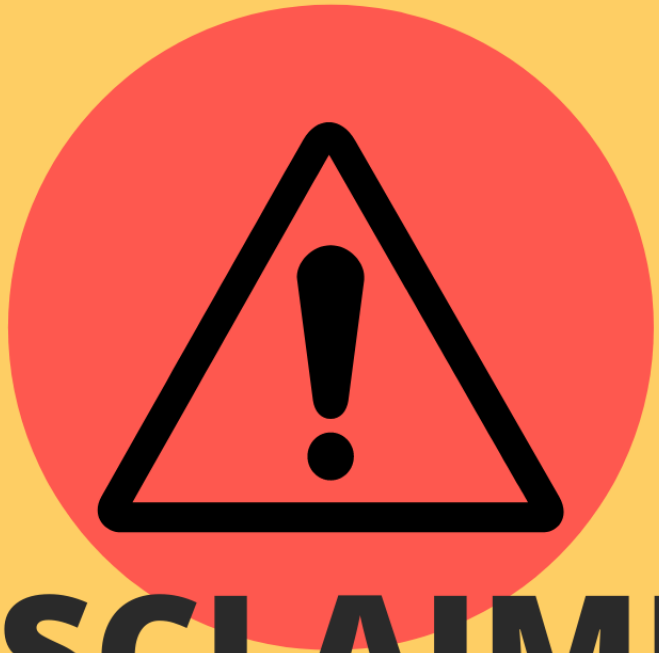

DISCLAIMER

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 VULNERABILITA' SOLIDITY

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

```
contract EtherGame {
    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
 }
```

RISPOSTA

Rappresenta un gioco semplice (che causerebbe situazioni di race conditions) in cui i giocatori inviano 0,5 Ether al contratto nella speranza di essere il giocatore che raggiunge per primo uno dei tre traguardi. Il primo a raggiungere il traguardo può rivendicare una parte dell'Ether accumulato quando il gioco è finito. Il gioco termina quando viene raggiunto il traguardo finale (10 Ether) e gli utenti possono richiedere i loro premi.

ERRORE: Il cattivo uso di this.balance nel codice può dare problemi se un malintenzionato potrebbe forzare il contratto mandando una piccola quantità di Ether tramite la funzione selfdestruct() per impedire a futuri giocatori di raggiungere un traguardo. In aggiunta, un attaccante più aggressivo potrebbe inviare forzatamente 10 Ether o più per spingere il saldo del contratto al di sopra del finalMileStone, così da bloccare per sempre tutti i premi nel contratto.

SOLUZIONE: La logica implementata nello smart contract, quando possibile, dovrebbe evitare di dipendere dai valori esatti del saldo del contratto perché può essere manipolato artificialmente. Aggiungere una nuova variabile, depositedWei per tenere traccia dell'Ether depositato. In questo modo non abbiamo più alcun riferimento a this.balance.

———Quale vulnerabilità è presente in questo smart contract in Solidity?

```
contract EtherStore {
```

```
        uint256 public withdrawalLimit = 1 ether;
        mapping(address => uint256) public lastWithdrawTime;
        mapping(address => uint256) public balances;

 function depositFunds() public payable {
       balances[msg.sender] += msg.value;
 }

 function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        require(_weiToWithdraw <= withdrawalLimit);
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
16      require(msg.sender.call.value(_weiToWithdraw)());
17      balances[msg.sender] -= _weiToWithdraw;
18      lastWithdrawTime[msg.sender] = now;
 }
}
```

RISPOSTA

L'attaccante può creare un nuovo contratto (diciamo all'indirizzo 0x0 ... 123) con l'indirizzo del contratto EtherStore come parametro del costruttore. In questo contratto sono presenti alcune funzioni, tra cui una anonima che si attiva nel momento in cui qualcuno mi da qualcosa. Se implementata tale funzione, tutte le call vanno in questa funzione, il contratto malizioso controlla se il suo saldo è maggiore di 1 eth, in caso positivo chiede di prelevare 1 eth da EtherStore. Il problema è che la call del contratto EtherStore non ha avuto ritorno, in quanto la funzione anonima non ha ancora dato un ritorno e nel frattempo ha richiamato la funzione di prelievo da EtherStore (su un altro thread). Questo scambio di call andrà avanti fin quando il saldo del contratto EtherStore sarà svuotato. Quando si esce dalla funzione anonima con ritorno, ci saranno aggiornamenti a cascata su EtherStore. (ATTACCO DI RIENTRO)

SOLUZIONE: Esistono numerose tecniche per evitare potenziali vulnerabilità di rientro negli smart contract.

1. Utilizzare la funzione transfer() quando si invia Ether a contratti esterni, che invia solo 2300 gas che non sono sufficienti per chiamare un altro contratto (ovvero reinserire il contratto di invio).
2. Assicurarsi che tutta la logica che cambia le variabili di stato avvenga prima che l'Ether venga inviato dal contratto. Nell'esempio EtherStore, le righe [17] e [18] di EtherStore.sol devono essere inserite prima della riga [16].
3. Introdurre un mutex, così da bloccare il contratto durante l'esecuzione del codice, impedendo le chiamate di rientro. (un mutex definisce una sezione critica). Definisco tanti mutex quanti sono le entry nella mappa

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

```
// library contract
contract FibonacciLib {
   uint public start ;
   uint public calculatedFibNumber ;
```

```
    function setStart ( uint _start ) public {
      start = _start ;
    }
    function setFibonacci ( uint n) public {
      calculatedFibNumber = fibonacci (n);
    }
    function fibonacci ( uint n) internal returns ( uint ) {
      if (n == 0) return start ;
      else if (n == 1) return start + 1;
      else return fibonacci (n - 1) + fibonacci (n - 2);
    }
}
contract FibonacciBalance {
    address public fibonacciLibrary ;
    uint public calculatedFibNumber ;
    uint public start = 3;
    uint public withdrawalCounter ;
    bytes4 constant fibSig = bytes4 ( sha3 (" setFibonacci ( uint256 )"));
    constructor ( address _fibonacciLibrary ) public payable {
      fibonacciLibrary = _fibonacciLibrary ;
    }
    function withdraw () {
      withdrawalCounter += 1;
      require ( fibonacciLibrary . delegatecall ( fibSig , withdrawalCounter ));
      msg . sender . transfer ( calculatedFibNumber * 1 ether );
    }
    function () public {
      require ( fibonacciLibrary . delegatecall ( msg. data ));
    }
}
```

RISPOSTA

- Il primo è una libreria e genera la sequenza di Fibonacci, e offre una funzione per ottenere il numero di posizione ennesima nella successione
- Il secondo è un contratto e consente a un partecipante di ritirare l'Ether dal contratto, con la quantità pari al numero di Fibonacci corrispondente all'ordine di prelievo del partecipante.

ERRORE: Entrambi hanno la variabile start, e la funzione di fallback in FibonacciBalance consente di passare tutte le chiamate al contratto di libreria, e potenzialmente chiamare setStart(). Le variabili di stato o di archiviazione (variabili che persistono nelle singole transazioni) vengono inserite in slot dello stack in modo sequenziale man mano che vengono dichiarate nel contratto.

Il contratto FibonacciBalance consente agli utenti di chiamare tutte le funzioni di FibonacciLibrary tramite la funzione di fallback, tra cui la funzione setStart(). Tale funzione è richiamata mediante DELEGATE CALL (utile per modulare il codice degli smart contracts), che preserva il contesto del contratto chiamante, quindi questa funzione consente a chiunque di modificare o impostare il valore in slot[0], ovvero l'indirizzo per FibonacciLibrary. Pertanto, un utente malintenzionato potrebbe creare un contratto dannoso, e chiamare setStart() passando l'indirizzo del contratto dannoso. Questo farà sì che ogni volta che un utente chiama withdraw() o la funzione di fallback, verrà eseguito il contratto dannoso (che può rubare l'intero saldo del contratto).

SOLUZIONE: Per ovviare a questa vulnerabilità, Solidity fornisce la parola chiave library per l'implementazione dei contratti di libreria. Ciò garantisce che il contratto con library sia senza stato (per attenuare le complessità del contesto di archiviazione) e non autodistruttibile.

———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;
    }
}
```

RISPOSTA

La vulnerabilità principale di questo contratto consiste nel fatto che vi è una vulnerabilità legata al limitato intervallo di rappresentazione in EVM, ovvero una vulnerabilità di overflow in tal caso in cui un malintenzionato, vista la visibilità pubblica del locktime potrebbe sfruttarla per alterare il valore e fare in modo di ritornare a un valore positivo, raggiungere il tempo e prosciugare il salvadanaio.

SOLUZIONE: una possibile soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare o costruire librerie matematiche sicure come SafeMath di Open Zepplin.