



# Hands-on: Solidity Patterns



CBSOft 2018

Blockchain & Smart Contracts Crash Course

Henrique Rocha ([henrique.rocha@gmail.com](mailto:henrique.rocha@gmail.com))

<https://github.com/hscrocha/bc-tutorial-2018>

<https://hscrocha.github.io/bc-tutorial-2018/>

# Find the issue...

```
1 pragma solidity^0.4.24;
2 /**
3  * @title Token Bank Unsecured
4  * @author Henrique
5  */
6 contract Bank {
7     struct Client {uint id; string name; uint balance;}
8     mapping(uint=>Client) private clients;
9     address private owner;
10
11     constructor() public{ }
12
13
14
15     function addClient(uint id, string name, uint balance) public {
16         require(owner == msg.sender);
17         clients[id] = Client(id,name,balance);
18     }
19
20     function changeBalance(uint id, uint balance) public { }
21
22
23
24 } //end of contract
```

# Private... but not really

- Contrary to common programming languages, private modifier does not make a variable invisible.
- Private makes the variable only accessible to the current contract, **HOWEVER** its contents are still visible to anyone in the blockchain.
- Developers must ensure data privacy by themselves
  - Either by hashing information and/or using a secondary private database (and avoid storing sensible info in the blockchain).

# Private... using address

```
1  pragma solidity^0.4.24;
2  /**
3   * @title Token Bank
4   * @author Henrique
5   */
6  contract Bank {
7      struct Client {address add; uint balance;}
8      mapping(address=>Client) private clients;
9      address private owner;
10
11     constructor() public{
12         owner = msg.sender;
13     }
14
15     function addClient(address clientAddress, uint balance) public {
16         require(owner == msg.sender);
17         clients[clientAddress] = Client(clientAddress,balance);
18     }
19
20     function changeBalance(address clientAddress, uint balance) public {↔}
24 } //end of contract
```

# Caution on Randomness

- It is tricky to create a truly random number in Solidity (if you don't want people to cheat)
- If you use the current timestamp as a seed, miners can (and probably will) manipulate the timestamp to get an advantage.
- The miner will also have access to many environment states (e.g., blockhash) and also the contract's internal state. Therefore it is not a good idea to rely on those.
- There are “Random generator” patterns

# State Machine Pattern

- A common pattern in contracts is to make them act as a state machine.
- The contract has states and function calls can transition the contract into a different one.
- Some functions can become unavailable (or behave differently) depending on which state the contract is.

# State Machine

```
6 contract CrowdFund{
7     enum State { Open, Refund, Closed }
8     State currentState = State.Open;
9
10    constructor(string url, uint min, uint exp) public {
16
17    function contribute() public payable{
18        checkExpiration();
19        require(currentState == State.Open, "Cannot contribute to an expired Crowdfund.");
20        pledges[msg.sender] = msg.value;
21        totalraised += msg.value;
22    }
23    function refund() public {
29    function checkExpiration() public {
30        require(currentState == State.Open, "Contract is already expired");
31        if(now >= expiration){
32            if(totalraised >= minimumRequired){
33                currentState = State.Closed;
34                getFunds();
35            }
36            else{
37                currentState = State.Refund;
38            }
39        }
40    }
```

# Find the issue...

```
2 ▾ /**
3   * @title Ether Bank Unsecured
4   * @author Henrique
5   */
6 ▾ contract Bank{
7     mapping(address=>uint) private balances;
8
9 ▾     constructor() public {↔}
11
12 ▾     function deposit() public payable {
13         balances[msg.sender] += msg.value;
14     }
15
16 ▾     function withdraw() public {
17         if(msg.sender.call.value(balances[msg.sender])){
18             balances[msg.sender] = 0;
19         }
20
21 ▾     function checkBalance() public view returns(uint){↔}
24 } //end of contract
```






# Re-entrancy

- Any transfer of Ether hands the control to the other contract. At that point it is possible for the other contract to execute functions (even calling back the original contract before the current function is resolved).
  - When using “transfer” or “send” this is not serious because of the gas limitation.
  - Using “call” is a different story. In the last example, we could drain all Ether from the contract.
- Always update the state before calling external functions (checks-effects-interactions pattern)

# Exploiting the Issue

```
2 ▾ /**
3   * @title Exploit Re-entrancy
4   * @author Henrique
5   */
6 ▾ contract ExploitWeakness {
7     address private owner;
8
9     constructor() public {↔}
12 ▾ function drain() public{↔}
16
17 ▾ function withdraw(address bank_address) public {
18     require(owner == msg.sender);
19     Bank(bank_address).withdraw();
20 }
21
22 ▾ function() public payable {
23     Bank B = Bank(msg.sender);
24     uint balance = B.checkBalance();
25 ▾ if( address(B).balance >= balance ){
26     B.withdraw();
27 }
28 }
29 } //end of contract
```

# Safe Re-entrancy

```
2  /**
3   * @title Ether Bank Safe Re-entrancy
4   * @author Henrique
5   */
6  contract Bank{
7      mapping(address=>uint) private balances;
8
9      constructor() public {}
11
12     function deposit() public payable {}
15
16     function withdraw() public {
17         uint user_balance = balances[msg.sender];
18         balances[msg.sender] = 0;
19         msg.sender.transfer( user_balance );
20     }
21
22     function checkBalance() public view returns(uint){}
25 } //end of contract
```

# Another issue...

- Even if we protect our contract against re-entrancy, someone can still try to take advantage on it.
- Consider the following Auction contract


# Auction Unsafe

```
2 ▾ /**
3   * @title Auction Unsafe
4   * @author Henrique
5   */
6 ▾ contract Auction {
7     address owner;
8     address winner;
9     uint winning_bid;
10
11 ▸     constructor() public {↔}
12
13
14
15
16
17 ▾     function bid() public payable{
18         require(msg.value > winning_bid); //check
19         //effects
20         address oldwinner = winner;
21         uint oldbid = winning_bid;
22         winner = msg.sender;
23         winning_bid = msg.value;
24         //interactions
25         oldwinner.transfer( oldbid );
26     }
27 } //end of contract
```

# Risk on Sending Funds

- Seems counter-intuitive but sending funds right after an effect may not be the best way.
- There is the potential risk for someone to cause the transfer to fail on purpose and keep the contract from ever performing its function.
- In the last example, it would be impossible to “out-bid” the attacker (causing him to win the auction)
- Use the “Withdraw pattern” to avoid this risk (each account is responsible to withdraw his losing bids)

# Withdraw Pattern

```
6 contract Auction {
7     address owner;
8     address winner;
9     uint winning_bid;
10    mapping(address=>uint) losing_bids;
11
12    constructor() public {}
13
14
15
16
17
18    function bid() public payable{
19        require(msg.value > winning_bid); //check
20        //effects
21        losing_bids[winner] += winning_bid;
22        winner = msg.sender;
23        winning_bid = msg.value;
24    }
25
26    function withdraw() public {
27        uint lost_bids = losing_bids[msg.sender];
28        losing_bids[msg.sender] = 0;
29        msg.sender.transfer( lost_bids );
30    }
31 } //end of contract
```