P2P Systems and Blockchains
Spring 2019,
instructor: Laura Ricci
laura.ricci@unipi.it

Lesson 19:
CONTRACT EXAMPLES AND VULNERABILITIES: THE DAO

24/05/2019
AN EXAMPLE: OPEN AUCTION

• State variables

```
// Address of auction beneficiary
address public beneficiary;

// Auction end time in Unix time
uint public auctionEndTime;

// Current state of the auction.
address public highestBidder;
uint public highestBid;

// Allowed withdrawals of previous bids
mapping(address => uint) pendingReturns;

// Set to true at the end, disallows any change
bool ended;
```

• Events

```
event HighestBidIncreased(address bidder, uint amount);
event AuctionEnded(address winner, uint amount);
```
**CONTRACT CREATION**

```solidity
constructor(
    uint _biddingTime,
    address _beneficiary
) public {
    beneficiary = _beneficiary;
    auctionEndTime = now + _biddingTime;
}
```

- initialize beneficiary and auctionEndTime
- contract creation transaction will take arguments as inputs
function bid() public payable {

    require(now <= auctionEndTime, "Auction already ended.");

    require(msg.value > highestBid, "There already is a higher bid.");

    if (highestBid != 0) {
        pendingReturns[highestBidder] += highestBid;
    }

    highestBidder = msg.sender;
    highestBid = msg.value;
    emit HighestBidIncreased(msg.sender, msg.value);
}

- **payable** keyword allows Ether to be sent with message call
- check that auction is ongoing and new bid is highest bid
- if new bid is higher, add old highest bid to `pendingReturns` list
- emit event notifying change in highest bid
WITHDRAW LOSING BIDS

function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;

        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}

• set balance of withdrawer to zero
• if withdrawal fails, restore amount in pendingReturns and return false
• if withdrawal succeeds, return true
function auctionEnd() public {

    require(now >= auctionEndTime, "Auction not yet ended.");
    require(!ended, "auctionEnd has already been called.");

    ended = true;
    emit AuctionEnded(highestBidder, highestBid);

    beneficiary.transfer(highestBid);
}

• check that `auctionEndTime` has passed
• check that `auctionEnd` has not been called before
• emit event signaling end of auction
• transfer highest bid to beneficiary
A Ponzi scheme:

- investors think they’re investing in a real company that produces returns to its investors, with the promise of high rewards.

- in reality, the money generated from new investment is used to pay off the interest accrued to the old investors.

- Ponzi schemes can last for years before falling apart dramatically.

- they are named for their originator, Charles Ponzi, who famously ran such a scheme in the 1920s.

- another famous Ponzi scheme: run by Bernie Madoff
  - took in $20 billion in investments over a span of 48 years
  - used the continued investment to convince investors their holdings were worth $65 billion
pragma solidity ^0.4.18;

contract SimplePonzi {
    address public currentInvestor;
    uint public currentInvestment = 0;

    function () payable public {
        // new investments must be 10% greater than current
        uint minimumInvestment = currentInvestment * 11 / 10;
        require(msg.value > minimumInvestment);

        // document new investor
        address previousInvestor = currentInvestor;
        currentInvestor = msg.sender;
        currentInvestment = msg.value;

        // payout previous investor
        previousInvestor.send(msg.value);
    }
}
The contract has two variables

- **currentInvestor**
  - address of the most recent investor in the contract.
  - this address is the only one that hasn’t received a return on the investment
  - if no one ever tops their bid, they will be the one to lose their investment

- **currentInvestment**
  - the amount of the investment one stand to lose if there are no further investments
A SIMPLE PONZI SCHEME IN SOLIDITY

```solidity
function () payable public {
    // new investments must be 10% greater than current
    uint minimumInvestment = currentInvestment * 11 / 10;
    require(msg.value > minimumInvestment);
}
```

- the contract has a single fallback function,
  - the fallback function can be executed by sending ether directly to the deployed contract’s address

- any new investment must be at least 10% greater than the current investment or it will be rejected.
  - to guarantee a juicy return to the investors
  - your shenanigans are not welcome here!

- to calculate the minimum investment need multiply by 1.1.
  - no decimals in Solidity, so need to multiply by 11 and then divide by 10
A SIMPLE PONZI SCHEME IN SOLIDITY

```solidity
// document new investor
address previousInvestor = currentInvestor;
currentInvestor = msg.sender;
currentInvestment = msg.value;
```

- onboarding the new sucker
- keep a reference to the previous investor so to pay out him/her with the new investment
- use send to be safe. Send may fail and return false
  - ignore the return value
  - overwrite the address of the old investor with the new investor
  - the ether will remain in the contract and be unclaimable
contract GradualPonzi {
    address[] public investors;
    mapping (address => uint) public balances;
    uint public constant MINIMUM_INVESTMENT = 1e15;

    function GradualPonzi () public {
        investors.push(msg.sender);
    }

    function () public payable {
        require(msg.value >= MINIMUM_INVESTMENT);
        uint eachInvestorGets = msg.value / investors.length;
        for (uint i=0; i < investors.length; i++) {
            balances[investors[i]] += eachInvestorGets;
        }
        investors.push(msg.sender);
    }

    function withdraw () public {
        uint payout = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(payout);
    }
}
REALISTIC PONZI

- added a minimum investment to prevent freeloaders from sending a 0-value transaction to become an investor

- new investments are distributed evenly between investors
  - to avoid adding the complexity of tracking investor shares
  - no incentive to send more than the minimum payment
  - after the distribution is complete, the newest investor is added to the list of investors

- note that the creator gets the privilege of joining the Ponzi without having to send any ether.

- as the number of investors in the Ponzi increases, the return for an investor from each new investment decreases.
BLOCKCHAIN SECURITY AT DIFFERENT LEVELS

Layer 0: Network
Layer 1: Blockchain
Layer 2: Application

Time progression from left to right.
SECURITY BUG: REENTRANCY

• when a non-recursive function is invoked, cannot it be re-entered before its termination?
  • the atomicity and sequentiality of transactions may induce programmers to believe that it cannot be reentered
  • however, this is not always the case!

• an attacker may exploit the fallback mechanism to re-enter the caller function.

• this may result in
  • unexpected behaviours
  • loops of invocations which eventually consume all the gas

• the famous DAO (Decentralized Autonomous Organization) Attack, which caused a huge ether loss in June 2016, exploited this vulnerability
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
...
```

Wallet

```javascript
uint balance = 10;

function withdraw(){
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```javascript
uint balance = 10;

function withdraw() {
    if (balance > 0) {
        msg.sender.call.value(balance)();
    }
    balance = 0;
}
```

10 ether
SECURITY BUG: REENTRANCY

User Contract

```
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```
uint balance = 10;

function withdraw()
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
```
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```javascript
uint balance = 10;

function withdraw()
{
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```

Later...

withdraw() - 10 ether

no transfer
**SECURITY BUG: REENTRANCY**

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
```

Wallet

```javascript
uint balance = 10;

function withdraw() {
    if(balance > 0)
        msg.sender.call.value(balance)();
    balance = 0;
}
```

Later...

Can the user contract withdraw more than 10 ether?
SECURITY BUG: REENTRANCY

User Contract

```javascript
function moveBalance() {
    wallet.withdraw();
}
...
function () payable {
    // log payment
}
```

Wallet

```javascript
uint balance = 0;
function withdraw {
    if(balance > 0)
        msg.sender.call.value(balance); // calls the default "payable" function
    balance = 0;
    // balance is zeroed after ether transfer
}```
This vulnerability resides in the fact that function ping is not reentrant, i.e. it may misbehave if invoked before its termination.
DECENTRALIZED AUTONOMOUS ORGANIZATION (DAO)

- a term coined by Vitalik in 2013
- a (digital) form of investor-directed venture capital fund
- investors from anywhere in the world can buy DAO tokens in exchange of money (Ether)
- owners of DAO tokens can vote on potential projects, i.e. how to spend funds, by voting in proportion to tokens
- profits from the investments go back to the stakeholders
- the lack of a centralized authority reduced costs and in theory provided more control and access to the investors.

<table>
<thead>
<tr>
<th>Investor</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100 Tokens</td>
</tr>
<tr>
<td>Bob</td>
<td>10 Tokens</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>110 Tokens</strong></td>
</tr>
</tbody>
</table>
• Slock.it: a Germany company tied to Ethereum foundation

• Example use case:

1. AirBnB user submits payment to the Ethereum blockchain

2. Slock Home Server (Ethereum client) receives the transaction

3. Power switch connected to Home Server receives “unlock” command, unlocks the door
THE DAO SMART CONTRACT

• Slack.it launched the DAO
  • the largest crowdfunding in history, raising over $150 million from more than 11,000 members.
  • built the DAO as a custom fundraising tool

• “the DAO”: a complex Smart Contract with many features to allow companies to make proposals for funding
  • most of the curators were notable members of the Ethereum community.

• the DAO token holders (aka DAO investors) vote on the proposals.
  • if the proposal received a 20% quorum
    • the requested funds would be released into the whitelisted contractor's wallet address.
  • in order to allow investors to leave the organization, in case a proposal that they saw as damaging or of poor quality was accepted
    • DAO created an "exit door" known as the "split function".
• split function:
  • allowed users to reverse the process and to get back the Ether they sent to the DAO.
  • if somebody decided to split from the DAO
    • they would create their own "Child DAOs" and approve their proposal to send Ether to an address after a period of 28 days.

• the DAO launch went smoothly and proposals were created and voted on

• security issues were raised during the first weeks
  • a big community call for a moratorium
  • it was not implemented and most of the security issues were not addressed fast enough.
  • the main DAO bug is showed in the next slides
THE DAO BUG AT A GLIMPSE

- on 18th June 2016, members of the Ethereum community noticed that funds were being drained from the DAO
  - the overall ETH balance of the smart contract was going down
  - a total of 3.6m Ether (about $70M at the time) was drained by the hacker in the first few hours.
- the attack exploits vulnerability found in the splitting function.
  - the attacker/s withdrew Ether from the DAO smart contract multiple times using the same DAO tokens.
DAO: THE HARD FORK

- a soft fork was proposed first:
  - blacklisting all the transaction coming from the child DAO of the attacker

- then the Ethereum community decided for a **hard fork**
  - the main miners agreed to come back in the history before the attack transaction
  - they renounced to all the rewards gained after the attack
    - otherwise the Ethereum prize would have been collapsed

- Ethereum Classic
  - a set of miners considered this centralized event not suitable for a cryptocurrency
  - they went on mining on the chain containing the attack transaction, giving birth to the Ethereum Classic chain.
TIMELINE AND AFTERMATH OF THE DAO ATTACK

• June 12 2016: slock.it developers announce that a bug is found, but no funds at risk

• June 17 (Morning): attacker drains $50M over 24 hrs of the DAO’s Ether ($50M) over 24 hrs

  Attacker’s funds were trapped in a subcontract for 40 days (July 27)

• June 17 (Evening): Eth Foundation proposes a “Soft Fork” to freeze the funds

• June 28: researchers publish a flaw in the Soft Fork Proposal

• July 15 (Morning): Eth Foundation proposes a “Hard Fork” to recover funds

• July 15 (Evening): “Ethereum Classic” manifesto published on github

• July 19: “Hard Fork” moves funds from attacker’s contract to recovery contract

  Ethereum Classic blockchain survives and is traded on exchanges

  Both Ethereum and Ethereum Classic are both around, reached new peaks