Exercise 8

1. In the lecture, you learned about the oracle design pattern.

   (a) What is an oracle and what is it used for?

   Solution:
   Oracles are third-party services that collect data from web services and write the data via a special smart contract to the blockchain. They are used by smart contracts that rely on external data from the oracle for computation of future states.

   (b) Explain the difference between a synchronous and an asynchronous oracle.

   Solution:
   Synchronous oracles write the data regularly to the blockchain (e.g. every 100 blocks). Contracts accessing the data get it instantly, but the data might be outdated.
   Asynchronous oracles expect the requesting contract to provide a callback function, which the oracle calls with the data. The request to the oracle emits an event, which the oracle service listens for. Afterwards, the service calls the callback function with the current data. The requesting contract might have to wait shortly for the data as it takes two transactions. In return, they get the most up-to-date data.

   (c) What are advantages and disadvantages of oracles?

   Solution:
   Advantages
   - Enables data retrieval from external sources
   - Synchronous oracle: easy to use
   - Asynchronous oracle: live data
   - Can be used for varying purposes (randomness, stock data, weather data, ...)

   Disadvantages
   - Costly in terms of gas consumption
   - Dependence on a third party (data manipulation and availability)

   (d) The provider of a synchronous oracle has a daily budget of 10 USD. Every how many blocks can they update the value of the oracle with this budget? Assume average transaction costs. Use current values from Etherscan.

   Solution:

Using values from May 26, 2019:

- Total number of transactions: 763,094 (`https://etherscan.io/chart/tx`)
- Total transaction fees: 359.586246278930226406 ETH (`https://etherscan.io/chart/transactionfee`)
- Number of blocks per day: 6,428 (`https://etherscan.io/chart/blocks`)
- Price of one ETH: 249.29 USD (`https://etherscan.io/chart/etherprice`)

$$\frac{6428}{x} \cdot \frac{359.586246278930226406}{763094} \cdot 249.29 \text{ USD} \leq 10 \text{ USD}$$
$$75.51 \leq x$$

They can update the value every 76th block.

2. A popular application of smart contracts are voting systems. In this exercise, we investigate such a system, which has a very subtle but common bug. Consider the following smart contract:

```solidity
pragma solidity >=0.5.1 <0.6.0;

contract FlawedVoting {
  mapping (address => uint256) public remainingVotes;
  uint256[] public candidates;
  address owner;
  bool hasEnded = false;

  modifier notEnded() {
    require(!hasEnded);
    _;
  }

  constructor(uint256 amountOfCandidates) public {
    candidates.length = amountOfCandidates;
    owner = msg.sender;
  }

  function buyVotes() public payable notEnded {
    require(msg.value >= 1 ether);
    remainingVotes[msg.sender] += msg.value / 1e18;
    msg.sender.transfer(msg.value % 1e18);
  }

  function vote(uint256 _candidateID, uint256 _amountOfVotes) public notEnded {
    require(_candidateID < candidates.length);
    require(remainingVotes[msg.sender] - _amountOfVotes >= 0);
    remainingVotes[msg.sender] -= _amountOfVotes;
    candidates[_candidateID] += _amountOfVotes;
  }

  function payoutVotes(uint256 _amount) public notEnded {
    require(remainingVotes[msg.sender] >= _amount);
    msg.sender.transfer(_amount * 1e18);
    remainingVotes[msg.sender] -= _amount;
  }

  function endVoting() public notEnded {
    require(msg.sender == owner);
    hasEnded = true;
    msg.sender.transfer(address(this).balance);
  }

  function displayBalanceInEther() public view returns(uint256 balance) {
      uint balanceInEther = address(this).balance / 1e18;
      return balanceInEther;
  }
}
```

(a) Describe the (intended) basic functionality of this contract.

Solution:
The creator of the contract specifies the amount of candidates which can be voted for. The amount of votes for each candidate are stored in the `candidates` array. Now accounts can buy votes with the designated `buyVotes()` function. One vote costs one Ether and if an account sends an odd amount of Ether, they get a refund of the money they payed too much. The votes are stored in the `remainingVotes` array and there is no limit to how many votes one account can buy. An account with votes can then call the `vote()` function to give a certain amount of their remaining votes to one candidate. This function can be called multiple times to vote for multiple candidates. If an account bought more votes than they actually wanted to give to candidates, they can convert their votes back into Ether by calling the `payoutVotes()` function. Finally, the contract creator can end the auction at any time with the `endVoting()` function. Afterwards, none of the other functions can be called anymore and the creator gets all funds that were sent to the contract during the auction.

(b) What is the problem with the contract? You can assume that the contract creator is benign and does not e.g. end the voting prematurely. If you are having trouble to see the bug, copy the code into the Remix IDE and try calling some functions.

Solution:
Anyone can vote without having to buy votes because the check in line 27 is not working as intended. The check is supposed to verify that an account has enough votes. However, as `remainingVotes[msg.sender]` and `amountOfVotes` are both unsigned integer variables, the difference between them is also unsigned. Therefore, it can never be less than 0 and the check is always true. What is more, because the amount of votes are subtracted from the remaining votes in the next line, this variable can underflow and its value can be enormously large as a result. An attacker could use this circumstance to get a refund of the votes of all other accounts using the `payoutVotes()` function.

(c) Which Solidity idiom is commonly used to prevent this type of error?

Solution:
Over- and underflow bugs can be prevented by using the SafeMath library.

(d) Rewrite the respective code to make the contract work as intended. How exactly does the idiom prevent the error?

Solution:
After including the SafeMath library in the code and adding the line `using SafeMath for uint256;` at the beginning of the contract, line 27 can be rewritten as follows: `require(remainingVotes[msg.sender].sub(amountOfVotes) >= 0);` Of course, all other usages of mathematical operators in the contract should be rewritten as well.
Applying the SafeMath library fixes the bug because the `sub()` function throws if `amountOfVotes` is larger than `remainingVotes[msg.sender]`.