Public Repository:

All code in:

- *git clone* *https://github.com/Deivitto/ZKP.git*

User: Deivitto
Mail: info.dcm62@gmail.com

A. Conceptual Knowledge

Before programming, it is important to know these main concepts. You should be able to explain these concepts to a (smart) five-year-old.

Reference this video for more information.

1. What is a smart contract? How are they deployed? You should be able to describe how a smart contract is deployed and the necessary steps.

   Smart contracts are digital contracts, basically programs, that are stored on a blockchain where they interact with users or other contracts (addresses) when certain conditions are met. We can say that smart contracts' finality is to provide a servicio or function in the way of contract to the entities that interact with them.

   This programs can't be modified once they are deployed on the blockchain, they can only be removed with a specific suicidal command. The benefits of this can be easily found in the transparency of the code that shows us what they are doing, but it provides in the same way one big problem, in the security if the smart contract is not created in a proper way, with good practices and enough knowledge and enough testing before deployment.

2. What is gas? Why is gas optimization such a big focus when building smart contracts?

   Gas is a concept that refers to the fee required for a transaction or interaction within a contract in Ethereum blockchain. Each operation on the EVM for example, uses different amounts of Gas, indeed, this is one the reasons why the optimization of the code implies removing unnecessary operations that

would consume extra gas. If you don't use enough Gas, the transaction will be slower or even fail. So for making our contract more usable, it needs to be optimized on the gas amount required or we will be paying tons of extra Ethers.

Also it is important to say that Gas price is not static, it depends on Ether price, so this concept of Gas is similar to the gas on gas stations.

3. What is a hash? Why do people use hashing to hide information?

   A hash is the result of a hash function, which creates a unique identifier for the piece of content you sent to the function, creating no matter what the length of the content you sent to it, a unique ciphertext of a specific length. In a less technical way: you send a String, it converts the String into another what is created based on computation and it has a fixed length, no matter if you send an "a" or if you send "lorem ipsum blablabla" it will create a string with fixed length of 256 bits long for example.

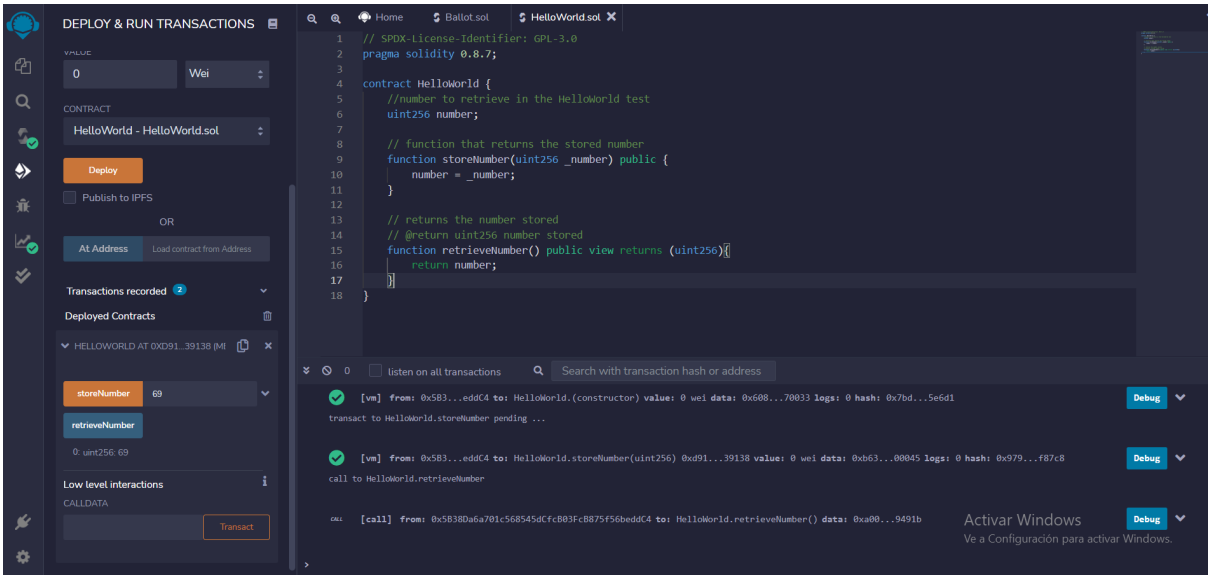
   In this way, rather than storing in a database the really secure password: "password" we will store a value like "ANX3BCOWPALSSX..." what means nothing in plain text. Then for checking the password, we don't need to know the stored password, we just need to perform the hash of "password" and check if it matches. Hashes used to be performed using concepts like salt and pepper, to increase the difficulty of finding the hash. Another typical usage is hashing multiple times, in that way, it is basically impossible to find the original key.

4. How would you prove to a colorblind person that two different colored objects are actually of different colors? You could check out Avi Wigderson talk about a similar problem here.

   In a simple way, I would take a photo in front of him of the 2 objects. I would send it to the computer, then I would check the HEX value in front of him with a color picker tool. By the way there are many ways to demonstrate the difference, but this is fast and logical.

B. You sure you’re solid with Solidity?

Hello World



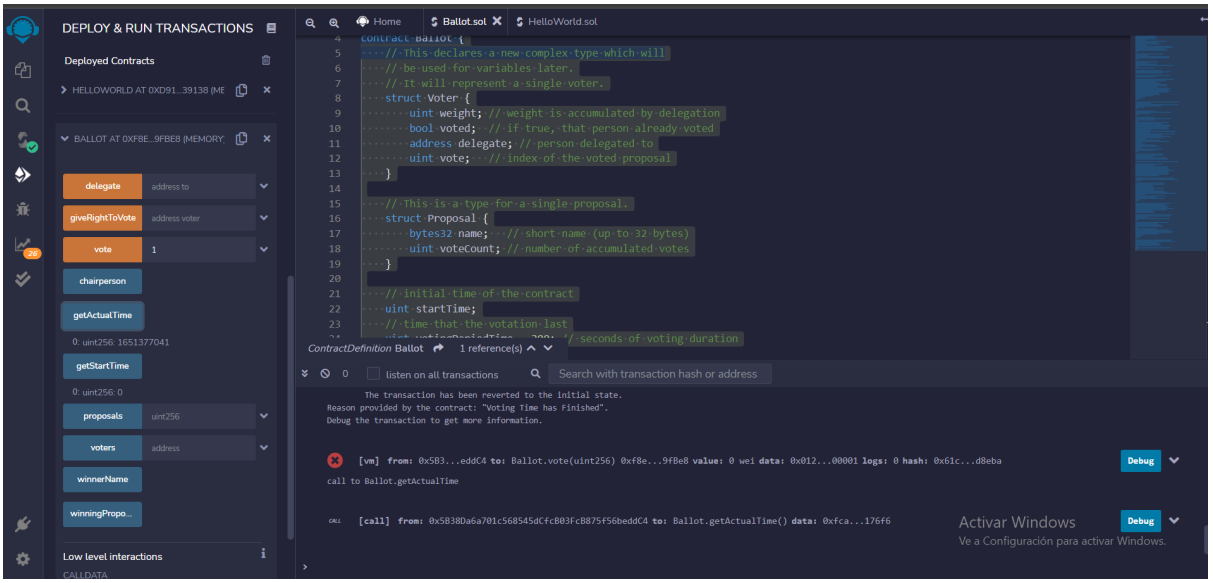

```sol
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.7;

contract HelloWorld {
    //number to retrieve in the HelloWorld test
    uint256 number;

    // function that returns the stored number
    function storeNumber(uint256 _number) public {
        number = _number;
    }

    // returns the number stored
    // @return uint256 number stored
    function retrieveNumber() public view returns (uint256){
        return number;
    }
}
```

Ballot



For showing the time, in case you don't use the JVM would be just that easy as using etherscan.io and see the txs.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    // initial time of the contract
    uint startTime;
    // time that the votation last
    uint votingPeriodTime = 300; // seconds of voting duration

    address public chairperson;
```

```
// This declares a state variable that
// stores a `Voter` struct for each possible address.
mapping(address => Voter) public voters;

// A dynamically-sized array of `Proposal` structs.
Proposal[] public proposals;

/// Create a new ballot to choose one of `proposalNames`.
constructor(bytes32[] memory proposalNames) {
    chairperson = msg.sender;
    voters[chairperson].weight = 1;
    // For each of the provided proposal names,
    // create a new proposal object and add it
    // to the end of the array.
    for (uint i = 0; i < proposalNames.length; i++) {
        // `Proposal({...})` creates a temporary
        // Proposal object and `proposals.push(...)`
        // appends it to the end of `proposals`.
        proposals.push(Proposal({
            name: proposalNames[i],
            voteCount: 0
        }));
    }
}

// modifier that revert the vote after finishing the voting period
modifier voteEnded {
        require(block.timestamp < (startTime + votingPeriodTime),
         "Voting Time has Finished");
        _;
}

// @return get starting time
function getStartTime() public view returns (uint) {
    return startTime;
}

// @return get actual time
function getActualTime() public view returns (uint) {
    return block.timestamp;
}
// Give `voter` the right to vote on this ballot.
```

```
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) external {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;
```

```
            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) external voteEnded {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
```

```
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```