Assignment 3

Blockchains & Distributed Ledgers
University of Edinburgh

Ábel Kocsis
S2115376

Monday 18th January, 2021

Contents

1 Part 1

At the beginning of my work, I did the following steps:

1. I have changed my MetaMask network to the Ropsten Test Network.
2. I have received some Ether to the test network. Unfortunately, the MetaMask's faucet has not worked for me, thus I used another faucet, which is available in this link.
3. As always, I had to change the environment of my Remix to Injected web3.
4. Then, I created a new file and pasted the code of the contract.
5. I had to modify the compiler version on Remix Ethereum to be able to compile the contract.
6. Then I could use the deployed contract at address `0x8B9CCE0437c421A53f033734772430A105BbBE89`

The task was to register my student number in the contract. In order to do so, I had to run the `register` function with two parameters: a `k` and my student id. As it can be seen in the first line of the `register` function, the hash of the `k` parameter must be the same as the hash of the `key` private variable of the contract. This practically means that `key` and `k` must be identical.

Since the default key was `this-is-the-key`, I have tried to use that as a key, which turned out to be not the current key. As it can be seen, the owner of the contract could modify the key using the `updateSeed` function. Since I am not the owner of the contract, I cannot do this. Thus, the current value of the `key` variable must be revealed.

The smart contract runs on the blockchain, thus, all variables are stored on the blockchain and can be seen. Checking local variables can be done using specific commands on the Remix Ethereum command line.[1] According to the article, I ran a command which is shown by Figure 1. The command does the following: it gets the Slot 1 (parameter 2) of the chosen contract (parameter 1), and prints it to the console (parameter 3). Slot 1 must be used in order to query the data of the key, since data is stored in the order of definition in the contract. The `owner` variable is in the first (Slot 0) slot, which takes a whole slot because of the `address` type. The `key` variable is the next one, thus is stored in the second slot (Slot 1).

```
web3.eth.getStorageAt("0x8B9CCE0437c421A53f033734772430A105BbBE89",
    1, console.log)
```

Figure 1: Getting the value of the private key. The command must be pasted in one line.

The result of the following command was `0x3b290000000000000000000000000000000000000000000000000000000000004`. This is the `byte32` encoding of the string, which must be decoded. An online tool[2] was used to decode the string, which was `;)`. Thus, this string was the `key` and had to be used as `k` parameter of the `register` function. With this, I was able to register my student number (S2115376).

1. *Accessing Private Data | Solidity by Example | 0.6.10*, accessed January 17, 2021, https://solidity-by-example.org/hacks/accessing-private-data/.
2. *Web3 Type Converter - Convert from bytes32 to string*, accessed January 17, 2021, https://web3-type-converter.onbrn.com/.

2 Part 2

2.1 Part 2a

2.1.1 High-level design decisions

My contract implements a so-called token bank, where users can buy, sell and transfer tokens and query their balances.

First of all, the `tokens` mapping maps each user (`address`) a number (`uint256`), which stores the current balance of the user. `uint256` type was used, since the number of the tokens held by a user cannot be negative, and can be assumed that a user can hold a lot of tokens. Since the number of tokens held by each user is stored in this mapping, the `getBalance` function simply returns with the proper value.

The `owner` variable (`address`) stores the address of the owner. It is set up during constructing the smart contract (`constructor`) and cannot be modified later.

The `tokenPrice` variable stores the current value of one token and the `allTokenNumber` variable stores the number of all tokens (both of them are `uint256`). The former was defined based on the task, while the importance of the latter is discussed below.

The **events** `Purchase`, `Transfer`, `Sell` and `Price` are emitted when some tokens are purchased, transferred, sold or the price of the tokens are changed, respectively.

Buying a token works as follows. The user must call the `buyToken` function, which checks if the right amount of weis (`tokenPrice * amount`) has been transferred. Then the `allTokenNumbers` variable is increased by the number of bought tokens and the tokens are added to the balance of the user (`tokens`). Lastly, the `Purchase` event is emitted and the function returns with a `true` value. If the requirement regarding the amount is false or overflow happens somewhere (increasing the number of tokens of the user or at all), the function is reverted.

Selling tokens works as follows. Firstly, the balance of the user is checked if he can sell this amount of tokens. If so, the balance (`tokens`) and number of all tokens (`allTokenNumbers`) are modified, the amount of money is transferred and the `Sell` event is emitted. Lastly, the function returns `true`. If the balance of the user is too low, the function is reverted.

Transferring the tokens works with the `transfer` function. It works almost in the same way as the selling function, except instead of transferring the money, the balance of the recipient is increased (`tokens`). In addition, the amount of all tokens is not modified. If overflow occurs, the whole function is reverted.

Changing the price of a token can be done by the owner of the contract using the `changePrice` function. Since the contract must have enough funds to make the users be able to sell all of their tokens, the account of the contract must be balanced. For example, if the owner doubles the fee of a token without adding money, only half of the existing tokens can be sold without an error. Thus, I decided to make the owner responsible for these funds. If the owner wants to increase the fee of tokens, he must spend as much money to the contract which is necessary to pay everyone in case of selling their tokens (`allTokenNumbers * newPrice - allTokenNumbers * oldPrice`). In the case of decreasing the price of tokens, the difference is sent to the owner, thus, he can use it to increase the fee later, and the contract will not hold unnecessary money. After these requirements are calculated, the new value of `tokenPrice` is set, the `Price` event is emitted and `true` is returned.

The `allTokenNumber` variable is used in order to make the calculation of price differences easy and fast during changing the price. In addition, iterating through the whole `address => uint256 token` mapping is not possible, thus this variable made available using this `mapping` structure as well. To make the owner be able to check how much money is needed when increasing the price, the `allTokenNumbers` variable is public.

In order to revert the functions when an overflow occurs, the OpenZeppelin's safe math library is used. Importing the library (`import "./SafeMath.sol";`) using another source file makes sure that the file is not modified after deploying the contract and the whole safe math library must not be copied into the TokenBank contract. The library can be used at any `uint 256` value since `using SafeMath for uint256;` was added to the contract. Thus, the addition (`add`), substraction (`sub`) and multiplication (`mul`) calculations are calculated using the specific functions.

The code of my contract can be seen at Appendix A.

2.1.2 Gas evaluation

Table 1 shows the transaction and execution costs of each public API of the contract. It is important to emphasize that since mapping is used, the increasing number of tokens and users does not increase the gas fees significantly.

	Deploy	Price	Balance	Change price	Buy	Sell	Transfer
Transaction	850604	22321	22352	33209	66116	22371	38162
Execution	609072	1049	1080	11745	44652	23278	30290

Table 1: Transaction and Execution cost of each function (gas). Price: `tokenPrice`, Balance: `getBalance`, Change price: `changePrice`, Buy: `buyToken`, Sell: `sellToken`, Transfer: `transferToken`

Different techniques were used to decrease the gas fees required by the contract. First of all, only necessary steps are programmed in the contract in the shortest and most efficient way. Secondly, the above-mentioned use of mapping (`tokens`) ensures that the gas costs will remain low even if the number of users increases. Thirdly, `require` functions were used instead of `assert` functions.[3] Lastly, the order of functions was determined how I predict the popularity of them. I assume, that the `getBalance` fucntion will be called most often and `changePrice` the least often. This also reduces the gas cost of the popular functions.[4]

3. Will Shahda, *Gas Optimization in Solidity Part I: Variables* [in en], August 2020, accessed January 17, 2021, https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde.

4. tak, *How to reduce gas cost in Solidity* [in en], February 2019, accessed January 17, 2021, https://medium.com/layerx/how-to-reduce-gas-cost-in-solidity-f2e5321e0395.

2.1.3 Potential hazards and vulnerabilities

First of all, in general, a similar TokenBank could be attacked by overflowing the tokens. If someone has a lot of tokens, an attacker could transfer them several tokens, which could cause an overflow, thus the victim could have fewer tokens after the transaction than before. This vulnerability can be mitigated by using the Safe Math library, which was used in terms of my contract as well.

Secondly, an attacker could use the property of SafeMath library to make others unable to use the contract. If a group of attackers (or a single attacker) would buy a huge amount of tokens which would increase the value of `allTokenNumbers` to $2^{256} - 1$, it would prevent anyone from buying new tokens. However, since this is a huge number of tokens, it is unlikely that anyone would buy this number of tokens just to prevent others from buying new ones. Also, the owner of the contract could mitigate by increasing the value of the tokens: the attackers might go out of money if they want to buy all of the available tokens.

Thirdly, this specification of a token bank needs trust in the owner. Since the owner could easily decide to decrease or increase the value of tokens, a buyer cannot predict the inflation of the tokens. Besides, the owner gets the difference if he decreases the value of tokens, which means that he might decrease the price of the tokens extremely after someone buy a lot of tokens. Thus, trust is necessary in the owner. A potential mitigation strategy could be that the smart contract allows the owner to modify the value of tokens, but only after a specific time with a specific amount. Let us say, the owner would be allowed to change the price only once per week and only increase or decrease it by a maximum of 20%. In this way, a rapid money loss would be mitigated.

To make the smart contract more secure, firstly, the sender of the contract is examined by the `msg.sender` and not by the `tx.origin` value. In this way, a potential vulnerability is mitigated since it is only the user who can call the functions. However, in this way, a third-party cannot be allowed to transfer funds from the account of the user. Secondly, at `sell` and `transfer` function, the substitution of the balance is done first and the money or token is only transferred after that.

2.1.4 Transaction history

2.1.5 Transaction history of using my contract

Table 2 shows the transaction history of testing my contract with a fellow student using the Ropsten Test Network. After both of us bought some tokens, I doubled the price. Then we bought some tokens again. After that, my fellow student transferred some tokens to me, which was followed by selling all of our tokens.

2.1.6 Transaction history of using another contract

Table 3 shows relevant addresses and history ids of an interaction with a contract of my fellow student.

Contract's address	0xC2870F94b2f761C27550ea2733e99C9EbE6Fc2d4
Own address	0xFd0E717891e8e16bba4b6EB0511833764683E477
Fellow student's address	0xFC77586810a83C6cEb4EAc195fA9E07022119AB7
buyToken - me	0x12d6279d293b7a6b5f98afa8753d7fcbbbbd295f69681925698dfccab0298b5a
buyToken - fellow student	0x7d83beb586f819ccb66a01617f63359424c4cf8296c47e6893d2e4d022d0e1fc
changePrice (double)	0xcd9ae1df610abd91f8d35b41f90efca8f89c067d4779abd688572b8ab6212e3d
buyToken - me	0x9123d7ca3a42a742069435393289228894 7a8fc451cfa9c172b6c526cb89cdeb
buyToken - fellow student	0x834e085e830ba73ce53d42188b8ab7f800715dc902d1d752aec44e80b4918d2b
transfer - fellow student	0xf4c9ad0bc7586f3252e6f247b6ff5a5fa1ae4cbc1c45fb0ed85bda6358fb03ea
sell - fellow student	0x4a54572b79674f94dbcbe6a5929fbc75afb763bff7b35ed0e7ff41f6dbb8064d
sell - me	0xb7329aaf1abebf17e6681307c8dd8663f16d3db540b3ae21dff3c9a67093a6a3

Table 2: Relevant addresses and transaction history ids of an interaction of my contract

Contract's address	0xB0dA6D81bf2E4Ec12bfc4a0c8cB008707b77f7c7
Own address	0xFd0E717891e8e16bba4b6EB0511833764683E477
Address of fellow student	0xFC77586810a83C6cEb4EAc195fA9E07022119AB7
buyToken	0x06738bd7156c0e96e62c2426d161f9af5c3127e3706a6871c4ef06b8290587bf
buyToken	0xf1bb4d1bf50eb72956847996b4349d037ca90131369eddb030be4807577cacf1
transfer	0x20af2d19dd9f47563da88c308942b01f70c143bb468f4bece312316ef05cf60e
sell	0x4c08fdcdaba33ef0ce2dac9746e68f18d9aa2767a0cc1c394f4a4dba65f53cd0

Table 3: Relevant addresses and transaction history ids of my interactions with the contract of my fellow student

2.2 Part 2b

2.2.1 Most basic approach

One of the simplest forms of a public-key directory is the Static Permissioned Blockchain.[5] Similar to that, we could require users to identify themselves. After that, during deploying the contract, the addresses of valid (identified) users could be ”hardcoded”, which could not be modified later. Thus, only these users should be allowed to buy new tokens. However, in this way, new users could not be added later, thus only a small amount of users could use the contract.

2.2.2 Verification by sending an encrypted text

Another way to authenticate users is if each user sends a public key with their documents. Then this public key can be added to the list of verified users. The buying process could be

5. *Blockchain & Distributed Ledgers Lectures*, Lecure 6, Slide 11.

the following: a user sends a message about his willingness to buy tokens, then the contract sends him a text encrypted by his public key. This text can be decrypted with only the private key of the user,[6] thus only the real user should be able to do it. Then the user can buy a token by sending the correct encrypted text as well as the necessary amount of money.

However, this process raises different questions. First of all, the encrypted text by the contract should be random. If the text wasn't random, anyone could predict it, thus anyone could claim that he is the user. However, this randomness is hard to reach in the blockchain. There are ways to generate random numbers which look random but can be easily recalculated (i.e. using the hash of the block or the time). Thus, a way to compute secure and private random number should be found.

In addition, the contract should remember the generated text in order to be able to verify if the user sends the correct decryption back. This text cannot be stored in the blockchain. There are different methods to store data off-chain in an Oracle[7] or with IPFS,[8][9] however, these methods usually aim to provide the information to all of the contracts in order to verify it, which is clearly a vulenrability at this case. Another way to remember the sent text without storing it on the blockchain is to store the hash of it.[10] In this way, the hash of the received encrypted text should be compared with the saved hash. However, it may be possible that a node could monitor the stack of the contract during generating the hash. In this way, the attacker could claim that he is the user.

Thus, this solution raises different security issues and cannot be used.

2.2.3 Verification with signature

Another solution idea is if the user sends his public key with their documents, which is then stored in the blockchain. Then, every token buy function must be signed by his private key. This signature can be verified by the public key[11].[12] This solution looks the most promising since the verification only requires the public key, and the signature does not reveal the private key of the user.

Thus, the process can be done using only on the chain with a signing and verification method.

6. *Public Key Encryption - Tutorialspoint*, accessed January 17, 2021, https://www.tutorialspoint.com/cryptography/public_key_encryption.htm.

7. *How to create your own Oracle with an Ethereum smart contract explained - step-by-step beginners guides | QuikNode*, accessed January 17, 2021, https://www.quiknode.io/guides/solidity/how-to-create-your-own-oracle-with-an-ethereum-smart-contract.

8. Adil H, *Off-Chain Data Storage: Ethereum & IPFS* [in en], October 2017, accessed January 17, 2021, https://didil.medium.com/off-chain-data-storage-ethereum-ipfs-570e030432cf.

9. Prince Sinha and Ayush Kaul, "Decentralized KYC System," *International Research Journal of Engineering and Technology (IRJET)* 5, no. 8 (2018): 1209–1210.

10. *How to secure Sensitive data on an Ethereum Smart contract? | by TALO Official | talo-protocol | Medium*, accessed January 17, 2021, https://medium.com/talo-protocol/how-to-secure-sensitive-data-on-an-ethereum-smart-contract-77f21c2b49f5.

11. __Sander, *Public / Private Keys and Signing* [in en-US], February 2018, accessed January 17, 2021, https://blog.todotnet.com/2018/02/public-private-keys-and-signing/.

12. *Blockchain & Distributed Ledgers Lectures*, Lecutre 8, Slide 19.

References

__Sander. *Public / Private Keys and Signing* [in en-US], February 2018. Accessed January 17, 2021. https://blog.todotnet.com/2018/02/public-private-keys-and-signing/.

Accessing Private Data | Solidity by Example | 0.6.10. Accessed January 17, 2021. https://solidity-by-example.org/hacks/accessing-private-data/.

Blockchain & Distributed Ledgers Lectures.

OpenZeppelin/openzeppelin-contracts [in en]. Accessed January 17, 2021. https://github.com/OpenZeppelin/openzeppelin-contracts.

H, Adil. *Off-Chain Data Storage: Ethereum & IPFS* [in en], October 2017. Accessed January 17, 2021. https://didil.medium.com/off-chain-data-storage-ethereum-ipfs-570e030432cf.

How to create your own Oracle with an Ethereum smart contract explained - step-by-step beginners guides | QuikNode. Accessed January 17, 2021. https://www.quiknode.io/guides/solidity/how-to-create-your-own-oracle-with-an-ethereum-smart-contract.

How to secure Sensitive data on an Ethereum Smart contract? | by TALO Official | talo-protocol | Medium. Accessed January 17, 2021. https://medium.com/talo-protocol/how-to-secure-sensitive-data-on-an-ethereum-smart-contract-77f21c2b49f5.

Public Key Encryption - Tutorialspoint. Accessed January 17, 2021. https://www.tutorialspoint.com/cryptography/public_key_encryption.htm.

Shahda, Will. *Gas Optimization in Solidity Part I: Variables* [in en], August 2020. Accessed January 17, 2021. https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde.

Sinha, Prince, and Ayush Kaul. “Decentralized KYC System.” *International Research Journal of Engineering and Technology (IRJET)* 5, no. 8 (2018): 1209–1210.

tak. *How to reduce gas cost in Solidity* [in en], February 2019. Accessed January 17, 2021. https://medium.com/layerx/how-to-reduce-gas-cost-in-solidity-f2e5321e0395.

Web3 Type Converter - Convert from bytes32 to string. Accessed January 17, 2021. https://web3-type-converter.onbrn.com/.

A Appendix

The code of my contract can be seen at Figure 2. The code of SafeMath can be found online.[13]

13. *OpenZeppelin/openzeppelin-contracts* [in en], accessed January 17, 2021, https://github.com/OpenZeppelin/openzeppelin-contracts.

```solidity
pragma solidity >=0.4.22 <0.7.0;
import "./SafeMath.sol";

contract TokenBank {
    using SafeMath for uint256;

    uint256 public tokenPrice = 10000;
    uint256 public allTokenNumber;
    mapping(address => uint256) tokens;
    address owner;

    event Purchase(address buyer, uint256 amount);
    event Transfer(address sender, address receiver, uint256 amount);
    event Sell(address seller, uint256 amount);
    event Price(uint256 price);

    constructor() public {
        owner = msg.sender;
    }

    function getBalance() public view returns (uint256){
        return tokens[msg.sender];
    }

    function buyToken(uint256 amount) payable public returns (bool) {
        require(msg.value == amount.mul(tokenPrice));
        allTokenNumber = allTokenNumber.add(amount);
        tokens[msg.sender] = tokens[msg.sender].add(amount);
        emit Purchase(msg.sender, amount);
        return true;
    }

    function transfer(address recipient, uint256 amount) public returns (
        bool) {
        require(tokens[msg.sender] >= amount);
        tokens[msg.sender] = tokens[msg.sender].sub(amount);
        tokens[recipient] = tokens[recipient].add(amount);
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    function sellToken(uint256 amount) public returns (bool){
        require(tokens[msg.sender] >= amount);
        allTokenNumber = allTokenNumber.sub(amount);
        tokens[msg.sender] = tokens[msg.sender].sub(amount);
        msg.sender.transfer(amount.mul(tokenPrice));
        emit Sell(msg.sender, amount);
        return true;
    }

    function changePrice(uint256 price) payable public returns (bool){
        require(msg.sender == owner);
        if (price > tokenPrice){
            uint256 diff = allTokenNumber.mul(price).sub(allTokenNumber.mul
                (tokenPrice));
            require(msg.value == diff);
        }
        else{
            msg.sender.transfer(allTokenNumber.mul(tokenPrice).sub(
                allTokenNumber.mul(price)));
        }
        tokenPrice = price;
        emit Price(price);
        return true;
    }
}
```

Figure 2: Code of my contract