Assignment 4

Blockchains & Distributed Ledgers
University of Edinburgh

Ábel Kocsis
S2115376

Monday 18th January, 2021

Contents

1 Fair Swap contract

This report shows a way how a fair swap can be conducted between tokens, where the token banks (smart contracts which enable users to deal with their tokens) are developed in a way how it was determined in Assignment 3.

1.1 General observations

First of all, I would like to briefly introduce how my solution was developing. The method of the investigation of the sender in the token bank (contract of assignment 3) has to be examined. This method could have a high impact on this contract as well, which is discussed below.

In general, a contract can call another contract in two different ways: using `call` and `delegatecall`. At first view, the usage of `delegatecall` looks promising since the `msg.sender` value in terms of this call is the original sender,[1] while in terms of the `call` function, it is the address of the (calling) contract. Thus, with `delegatecall`, it is possible to make the Token Bank contract believe that the actual sender is the user even if the caller is the FairSwap contract.

However, in terms of `delegatecall`, the calculations run on the storage of the original contract, thus on the memory of the Fair Swap contract.[2][3] In this way, the state of the other contract (Token Bank) cannot be modified.

If the Fair Swap contract uses the `call` function to call the Token Bank contract, the `msg.sender` value is the address of the Fair Swap contract. Thus, the contract is not allowed to transfer any money from the balance of the users since there is no delegate API in the Token Bank function.

A way to deal with this problem is to modify the Token Bank contract to examine the `tx.origin` of the sender when investigating the sender of a `transaction` call. However, first, it would be cheating because the assignment 4 does not allow us to modify the contract. Second, it would not solve the problem because there is no way to conditionally delay calls, thus, after the users promised each other to make a swap through the FairSwap contract, both of them should call the FairSwap contract in order to let it done the transaction. In this way, the second user could easily 'disappear' after the first transaction.

Thus, the Token Bank contract will continue examining the `msg.sender` of the messages, and another way will be developed to ensure the fair swap.

1. zeroFruit, *DelegateCall: Calling Another Contract Function in Solidity* [in en], August 2020, accessed January 17, 2021, https://medium.com/coinmonks/delegatecall-calling-another-contract-function-in-solidity-b579f804178c.
2. *Ethernaut Lvl 6 Delegation Walkthrough: How to abuse the delicate delegatecall | by Nicole Zhu | Coinmonks | Medium*, accessed January 17, 2021, https://medium.com/coinmonks/ethernaut-lvl-6-walkthrough-how-to-abuse-the-delicate-delegatecall-466b26c429e4.
3. *Blockchain & Distributed Ledgers Lectures*, Lecture 4, Slide 30.

1.2 How to use the contract

This subsection provides a brief summary of how to use my Fair Swap contract. The reasons for the design decisions is discussed in the next subsection.

1. The users have to agree (in the real world) on the parameters of the swap and need to provide each detail to each other: address of contracts, amount of swapping tokens and the addresses of each other.
2. One of the parties (Party 1) calls the `book` function of the Fair Swap contract. If it is reverted, they have to wait because the contract is currently used. If it is not reverted, they can continue the process.
3. Party 1 transfers (using the `transfer` function of the Token Bank contract) the tokens which will be swapped to the address of the Fair Swap contract (not to the other user!). Now, the Fair Swap contract owns the tokens, which are intended to swap by Party 1.
4. Party 1 calls the `transactionDone` function of the Fair Swap contract which requires 1 ETH and the following parameters
   - address of contract where his tokens are stored (`ownContract`)
   - address of contract where the tokens of the other party are stored (`oppContract`)
   - address of the other party (`oppAddr`)
   - amount he will send (has sent) during the swap (`ownAmount`)
   - amount he will receive during the swap (`oppAmount`)

   If Party 1 has done everything properly, the contract validates that it has received the tokens.
5. Party 2 calls the `check` function of Fair Swap contract with the following parameters
   - address of contract where his tokens are stored (`ownContract`)
   - address of contract where the tokens of the other party are stored (`oppContract`)
   - address of the other party (`oppAddr`)
   - amount he will send during the swap (`ownAmount`)
   - amount he will receive during the swap (`oppAmount`)

   If the function is reverted, the opponent has tried to cheat and Party 2 should stop all movements. Otherwise the procedure continues.
6. Party 2 transfers (using the `transfer` function of the Token Bank contract) the tokens which will be swapped to the address of the Fair Swap contract (not to the other user!). Now the contract owns all the tokens which will be swapped.
7. Party 2 calls the `transactionDone` function of the Fair Swap contract, sends it 1 ETH and input the parameters, which are the same as above (but in the other view). The contract validates that Party 2 has done the transaction properly.
8. If everything was done properly, the tokens are swapped and transferred to the balance of users. Both player receive some money back regarding their gas usage in order to ensure the fairness of the contract.

In the case of Party 2 disappears, Party 1 can call the `withdraw` function with his contract address as parameter which deletes the process and sends him back his tokens. Please, note that the name of 'own contract' refers to the contract where the tokens of the user are stored, but this contract do not have to be owned by the user.

1.3 Design description

In general, users are required to transfer their tokens to the address of the contract, which then does the fair swap of tokens. In this way, no one is required to send his token to the other party in advance who could then disappear. The contract is designed to ensure security as well as fairness, which is presented in this subsection.

1.3.1 Variables

The following variables are defined in the Fair Swap contract:

- `party1`, `party2` (`address`): The addresses of the parties participating in the swap. This contract supports only one swap at a time.
- `contract1`, `contract2` (`address`): `contracti` is the address of Token Bank contract where the tokens of Party i are stored, which he would like to swap. The two contract can be identical.
- `amount1`, `amount2` (`uint256`): The amounts they would like to swap.
- `p1Gas`, `p2Gas` (`uint256`): the gas used by the users.
- `startTime` (`uint`): the time when the last function call was done. Used for examining timeout
- `status` (`uint8`): Status of the contract. Status 0: the contract is free. Status 1: the contract is booked. Status 2: the contract is booked and Party 1 has already proven to sent his tokens to the contract.

To put this variables in context: `Party1` will swap `amount1` of his tokens in the Token Bank at address `contract1` with `amount2` of the tokens of `Party2` at the Token Bank at address `contract2`.

1.3.2 Book function

One of the key challenges of Fair Swap is that it is not possible to monitor the sender of the tokens. Without a booking function, the following scenario can happen: Party X would like to swap tokens with Party Y, while Party A would like to swap tokens with Party B. Party X and Party A uses the same Token Bank contract and both of them send their tokens to the Fair Swap contract in advance. Then, both of them would like to notify the contract he has done

the transaction, which cannot decide whose transaction was really done. In a worse case, an attacker could monitor the transaction of Party A. After Party A transacted any tokens to the Fair Swap contract, the attacker could call the `transactionDone` function claiming that these were his tokens and the contract could no distinguish between the tokens of the attacker and Party A.

Thus, one of the parties (Party 1) is required to call the `book` function of the contract in order to mitigate this scenario. After the booking process succeeded, Party 1 can be sure that the Fair Swap contract is reserved and he can start transferring his tokens.

The `book` function is reverted if the status of the contract is not `0`, thus someone uses the contract. Otherwise, it saves the address of the caller and resets the variables (`startTime` and gas variables). If there is no activity for 180 seconds, a new booking is allowed, when the previous one is deleted. Please, note that this timeout should be fine-tuned to ensure usability as well as mitigate long waiting queues.

1.3.3 Transaction done function

After transferring the tokens to the address of the FairSwap contract, users are required to call the `transactionDone` function. The contract cannot check if tokens were sent by the user. However, what it does is to check if the necessary amount of tokens are in its balance or not. If its balance is less than the amount what should have been transferred by the party, the party clearly has not done the transaction. On the other hand, no one should have transferred other than the user, thus, if the balance is at least the amount which was claimed by the party, it is assumed that it was sent by the user.

The function is designed to accept the user whenever the balance of the contract is larger or equal than what is claimed by the user. The reason for this is to mitigate the following scenario. If the function were to check if the balance equal to the necessary amount, an attacker could just send a token to the contract, and the swap of the honest parties would be prevented. That is why a higher amount of balance is allowed.

Apart from checking the balance of the contract different steps are done in this function. When Party 1 calls the `transactionDone` function, the details of the swap, which are required by the function as parameters, are saved (the description of the parameters can be seen above). The status of the contract is set to 2. The timeout is also updated, thus Party 2 has 1 minutes to do the transactions and call the function.

When Party 2 calls the function, the details of the swap are checked. It is discussed later why transferring the tokens and calling this function is safe for Party 2. After the balance, as well as the parameters, are checked, the transfers happen automatically: Party 1 receives the tokens of Party 2 and vice versa. The result of the `transfer` calls are checked, however, it cannot happen that the transactions fail because of insufficient funds since the balance of the contracts has already been checked.

Another corner case, which is worth to examine is when the two parties would like to change tokens in the same contract, thus `contract1` and `contract2` are identical. At that case, after the transaction of Party 2, it has to be checked if the balance of the contract in that Token Bank is not less than the sum of the two amounts.

When users call the `transactionDone` function, they are required to pay 1 ETH to the contract. After the swap was done, they got back some of their money, modified by values in order to ensure the fairness of the contract. This is expanded later on subection 1.5.

1.3.4 Check function

As we could see, Party 1 can be sure that he could start his transaction after the booking since no one else should do the same. However, what about Party 2? As it was mentioned, the `transactionDone` function does not succeed in case of different parameters of the two parties. However, Party 2 is required to transfer her tokens before calling the `transactionDone` function. Thus, in case of a dishonest Party 1, the swap does not occur, but Party 2 has already transferred her tokens, which is on the balance of the contract. A `withdraw` function could solve the issue, however, we would like to ensure that Party 2 can be sure about the truthfulness of Party 1 before transferring any token.

Thus, the `check` function is created, which aims to check if Party 1 has already transferred his tokens (`status == 2`) as well as used the same parameters which had been agreed with Party 2. If not, the function is reverted and Party 2 can leave the Fair Swap contract without transferring any tokens. If so, Party 2 knows that Party 1 has already properly done everything, thus, he is waiting for Party 2 to transact the tokens and call the `transactionDone` function.

1.3.5 Withdraw function

It was already discussed what if Party 1 is dishonest. However, it is also possible that Party 2 has the intention to cheat which should be mitigated.

The `transactionDone` function ensures that the swap does not succeed if Party 2 has not transferred the right amount of money or used other parameters as Party 1. However, Party 2 could decide to do nothing, which would mean that the tokens of Party 1 remain in the balance of the contract. In this way, the tokens of Party 2 would not change, while Party 1 would lose a certain amount of tokens.

That is why a `withdraw` function is created. This function enables Party 1 to cancel its booking if the transaction was not done yet when the status of the contract is 1. It also enables Party 1 to withdraw from the transaction if he had already transferred his tokens and called the `transactionDone` function, and the 60 seconds timeout is reached without the movement of Party 2. In this way, Party 1 receives back all of his tokens as well as the transferred 1 Ether.

It could also happen that a user transfers some tokens accidentally. For that case, if the status of the contract is 0, thus nobody reserved the contract, one could ask for the tokens which are held by the contract at a specific Token Bank contract. This is enabled since firstly, the Fair Swap contract should not hold any tokens at stage 0, thus this tokens should be there accidentally. Secondly, there should be a way how these tokens can remain. However, I would like to emphasise that these tokens can be claimed by anyone, nut just by who transferred them.

It could be also asked if it is possible to further develop the withdraw function in a way how

further privileges could be added such as claiming tokens by Party 2 from his contract or by anyone else during status 1 from a random contract. However, I would argue that these lines would make the logic of the contract unnecessarily difficult, which would lead to harder understandability as well as possible errors. Thus, at this moment, only the necessary withdrawing methods are implemented, and users are required to be responsible and only transfer tokens when it is suggested.

1.4 Gas analysis

The gas costs of the contract can be seen at Table 1. Different techniques have been used in order to reduce the gas fees of the contract. To begin with, only necessary steps are taken in each function and `call` functions are used instead of importing the whole code of the contracts. Moreover, the usage of `require` functions instead of `assert` functions reduce the gas costs.[4] Besides, the code starts with the most popular functions (`book`, `transactionDone`), which also reduces the costs of these functions.[5]

	Deploying	**Book**	**P1 - transDone**	**check**	**P2 - transDone**
Transaction cost	2024273	104867	150665	53253	108054
Execution cost	1519789	83595	124785	27373	112174

Table 1: Transaction and Execution costs of the functions (gas). transDone: transactionDone

1.5 Fairness

As it can be seen at Table 1, the costs of the functions can vary. In addition, since the final swap transactions are called after Party 2 calls the `transactionDone` function, in case of an expensive `transfer` function (which can be anything which follows the predetermined API), the costs of Party 2 can rocket. This scenario is clearly unfair, which should be mitigated.

That is why the costs of the `book`, `transactionDone` and `check` functions are monitored and saved. The gas spent by Party 1 is stored in `p1Gas`, while the one spent by Party 1 is stored in `p2Gas` variable. During calling the `transactionDone` function, both of them are required to transfer 1 ETH to the contract. Thus, at the end of the swap (at the end of `transactionDone`), the difference between the gas costs are calculated. The party who has spent more on gas costs gets more than 1 ETH (`1 ether + p1Gas - p2Gas`), while the other one gets less (`1 ether - (p1Gas - p2Gas)`) in order to make their costs equal. Even if it is just an estimation and the real costs can be different from the estimated ones, the huge differences caused by the call of `transaction` functions can be mitigated with this strategy.

It is also important to emphasise that requiring the users to send 1 ether to the contract regarding the gas fees might be too much. It would disadvantage parties with less money (less than 1 ETH) and the gas fees might never go close to 1 ETH. However, I would argue that different transaction functions should be investigated in order to define an upper bound of the

4. 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.

5. 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.

differences of gas costs. At this moment, 1 ETH is clearly an upper bound which could be further examined.

1.6 Potential vulnerabilities

As it was stated above, different vulnerabilities such as the malicious Party 1 or Party 2 is mitigated with this contract. It should be also mentioned that since both users are required to use the address of their opponents as the parameter of the `transactionDone` function, nobody else can successfully call these function than the one who booked the contract and his given opponent.

A potential vulnerability of this contract is that it could be locked by a malicious party. Even if the `book` function can be called after 180 seconds of inactivity, an attacker could constantly call the `book` function preventing everybody else using the Fair Swap contract. This scenario could be solved with different strategies. For example, a waiting list could be constrained where the users can wait in a queue. After the first finished with the swap, the second one can start. If everyone could be at most once in the list, no one could lock the contract. In addition, a blacklist could be also added. A user who is on the blacklist should not be allowed to use the contract at all.

Another vulnerability of the contract is that the `check` function can be called numerous times. Since the gas costs are distributed, Party 2 could keep calling this function, and half of the costs of these calls should be paid by Party 1. Besides it is unlikely that Party 2 would like to do this (since half of the prices are still on him), we could ensure that this function can be called at most once.

A potential vulnerability of the contract is if users use it in a non-recommended way. Thus, if someone transfers tokens to the contract at status 0 without calling the `book` function, one could easily get his tokens by calling the `withdraw` function with the address of his contract. Besides, if someone transfers money to the contract during another swap (status 1 or 2), there is no way to withdraw it before the end of the swap, thus, the parties could easily claim these tokens. For example, Party 1 books the contract and Party X transfers 2 tokens in contract C. Then Party 1 can easily call `transactionDone` with his own address (even as the opponent address) and claim the tokens transferred by Party X. A potential mitigation strategy could be if the properties of the swap would be fixed at the time of booking. Thus, Party 1 could not change his mind about the number of tokens and the address of the contracts, which would decrease the probability that a random-arrived token is claimed by Party 1. The whole threat could be only mitigated if the tokens would have ids, and every token should be tainted in advance with the address of the owner. However, this is not possible with the current API of the token Banks.

Lastly, malicious Token Bank contracts are also hazards. Since there is no way to make the FairSwap contract analyse the other contract, it will call its transfer function without knowing what it does. To mitigate any vulnerability regarding this scenarios, both users are required to study both of the contracts, and begin the swapping if any only if both of the contracts are secure.

1.7 Transactions

Table 2 shows the relevant addresses and transaction ids of an interaction with my FairSwap contract, deployed in the Ropsten Test Network. The relevant transactions to the TokenBank contracts are not presented since the Fair Swap contract ensures that those transactions have been done.

FairSwap address	0x01155CB9b11E0DEA0a0607616a25cCb9595Ba583
My address	0xFd0E717891e8e16bba4b6EB0511833764683E477
Fellow student's address	0xFC77586810a83C6cEb4EAc195fA9E07022119AB7
My Token-Bank's address	0xC2870F94b2f761C27550ea2733e99C9EbE6Fc2d4
Fellow student's TokenBank address	0x29b540De788B72B2e5e5e18eFBd774Ef94BCb0A7
book - fellow student	0xe987a6b9d9f0e0a253c95057bf9d56188142e1332ab279cfc673eb831b2ddb70
transactionDone - fellow student	0x4b0d41ff3816b5d193015e2c605adaa59a868f22c2bbd73d847d0f95bb2f3b74
check - me	0x5ea0575e87ad53d25ced0140c2e17d8463d70f7f1080b516735f22bc87b3738f
transactionDone - me	0x8db0ceeea94a0fb488baab783826e586d273dbabdb183b42295e3518b29146a9

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

The TokenBank contract of the other student was slightly different from mine, which did not allow us to test using both of our contracts. Thus, two deployed version of my Token Bank contract were used: one deployed by me and one deployed by the fellow student. Of course, we communicated with the same FairSwap contract. All the transactions worked in the correct way.

My transactions using the contract of my fellow student is presented at Table 3.

Address of the "token bank" of the fellow student, deployed by me	0x33686837C5Cbd000dc917ec5Cf4Ca56b7481D156
SafeSwap address	0x2b232cA688650b2cD4d8F330caC51Dc201938ec8
authorize	0x605d3ae9168dc6da26f2f38aadef078a427af70ad6bf910fa2afcab4c8c677f5
swap	0x71320128ca3b5c315a1e9329d4735237e4dc1f5fa497b44eeb8bc521db6c8491

Table 3: My interactions with the contract of my fellow student and relevant addresses

References

Blockchain & Distributed Ledgers Lectures.

Ethernaut Lvl 6 Delegation Walkthrough: How to abuse the delicate delegatecall | by Nicole Zhu | Coinmonks | Medium. Accessed January 17, 2021. https://medium.com/coinmonks/ethernaut-lvl-6-walkthrough-how-to-abuse-the-delicate-delegatecall-466b26c429e4.

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.

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.

zeroFruit. *DelegateCall: Calling Another Contract Function in Solidity* [in en], August 2020. Accessed January 17, 2021. https://medium.com/coinmonks/delegatecall-calling-another-contract-function-in-solidity-b579f804178c.

A Appendix

Figure 1, Figure 2 and Figure 3 show the code of my contract.

```
1  pragma solidity >=0.4.22 <0.7.0;
2
3  contract FairSwap {
4
5      address payable party1;
6      address payable party2;
7      address contract1;
8      address contract2;
9      uint256 amount1;
10     uint256 amount2;
11     uint256 p1Gas; // gas used by party1
12     uint256 p2Gas; // gas used by party2
13     uint startTime;
14     uint8 status; // 0: no booking, 1: booked, 2: party1 has transferred
15
16     function book() public{
17         uint startGas = gasleft();
18         require(status == 0 || block.timestamp > startTime + 180, "Contract
               has already booked");
19         status = 1;
20         party1 = msg.sender;
21         startTime = block.timestamp;
22         p2Gas = 0;
23         p1Gas = startGas - gasleft();
24     }
```

Figure 1: Code of my contract - Part 1

```
25
26      function transactionDone(address ownContract, address oppContract,
            address payable oppAddr,  uint256 ownAmount,  uint256 oppAmount)
            payable public{
27          uint startGas = gasleft();
28          require(msg.value == 1 ether, "1 Ether must be sent to the contract
                ");
29          bool callStatus;
30          bytes memory result;
31          require(status < 3, "Status error");
32          if (status == 1){
33              require(party1 == msg.sender, "User error");
34              (callStatus, result) =  ownContract.call(abi.
                    encodeWithSignature("getBalance()"));
35              require(callStatus && abi.decode(result, (uint256)) >=
                    ownAmount, "getBalance error");
36              contract1 = ownContract;
37              party2 = oppAddr;
38              contract2 = oppContract;
39              amount1 = ownAmount;
40              amount2 = oppAmount;
41              status = 2;
42              startTime = block.timestamp;
43              p1Gas += startGas - gasleft();
44          }
45          else if (status == 2){
46              require(party1 == oppAddr && party2 == msg.sender && contract1
                    == oppContract && contract2 == ownContract
47                   && amount1 == oppAmount && amount2 == ownAmount, "Parameter
                         error");
48              (callStatus, result) =  ownContract.call(abi.
                    encodeWithSignature("getBalance()"));
49              require(callStatus, "getBalance error");
50              if (contract1 == contract2){
51                  require(amount1 + amount2 >= amount1 && abi.decode(result,
                        (uint256)) >= amount1 + amount2, "GetBalance amount
                        error");
52              }
53              else{
54                  require(abi.decode(result, (uint256)) >= amount2, "
                        getBalance amount error");
55              }
56
57              (callStatus, ) =  contract1.call(abi.encodeWithSignature("
                    transfer(address,uint256)", party2, amount1));
58              require(callStatus, "Contract1 transfer error");
59              (callStatus, ) =  contract2.call(abi.encodeWithSignature("
                    transfer(address,uint256)", party1, amount2));
60              require(callStatus, "Contract2 transfer error");
61              status = 0;
62              p2Gas += startGas - gasleft();
63              party1.transfer(1 ether + p1Gas - p2Gas);
64              party2.transfer(1 ether - (p1Gas - p2Gas));
65          }
66
67      }
```

Figure 2: Code of my contract - Part 2

```
68
69      function check(address ownContract, address oppContract, address
            payable oppAddr,  uint256 ownAmount,  uint256 oppAmount) public
            returns (bool){
70          uint startGas = gasleft();
71          require(status == 2 && party1 == oppAddr && party2 == msg.sender &&
                contract1 == oppContract && contract2 == ownContract
72              && amount1 == oppAmount && amount2 == ownAmount, "Parameter
                    error");
73          p2Gas += startGas - gasleft();
74          return true;
75      }
76
77      function withdraw(address contr) public{
78          bool callStatus;
79          bytes memory result;
80          if (status == 0){
81              (callStatus, result) =  contr.call(abi.encodeWithSignature("
                    getBalance()"));
82              require(callStatus, "getBalance error");
83              if (abi.decode(result, (uint256)) > 0){
84                  (callStatus, ) =  contr.call(abi.encodeWithSignature("
                        transfer(address,uint256)", msg.sender, abi.decode(
                        result, (uint256))));
85                  require(callStatus, "Transfer error");
86              }
87          }
88          else if (status == 1){
89              require(msg.sender == party1, "Only party1 is allowed to
                    withdraw");
90              status = 0;
91          }
92          else {
93              // status == 2
94              require(msg.sender == party1 && contr == contract1 && block.
                    timestamp > startTime + 60, "Only party1 is allowed to
                    withdraw");
95              (callStatus, ) =  contr.call(abi.encodeWithSignature("transfer(
                    address,uint256)", msg.sender, amount1));
96              require(callStatus, "Transfer error");
97              party1.transfer(1 ether);
98              status = 0;
99          }
100     }
101 }
```

Figure 3: Code of my contract - Part 3