Blockchains & Distributed Ledgers - Coursework 3

s1519734

April 18, 2025

1 Part 1

The contract's functionality is quite simple. The "register" function requires the correct key for it to work. The initial value of the key (at contract creation time) is "this-is-the-key"; however, it can be modified by the contract owner using the "updateSeed" function.

Using `https://ropsten.etherscan.io`, it can be seen that the contract was created in transaction "0xb7821757e6bbbee865426174200efb5a04fdb7d90e7b11635e672152e625e9ef". The transaction's Etherscan page indicates that the address of the contract owner is "0xc8e8add5c59df1b0b2f2386a4c4119aa1021e2ff". There are only two transactions initiated by the owner in the contract's transaction history, the first being the creation of the contract, naturally. As the contract code has been made public, Etherscan can once again be used to see what function was called in each transaction and decode the inputs to those functions (first 4 bytes can be used to decipher which method was called and then each input takes up 32 bytes, see: `https://www.trustology.io/insights-events/decoding-an-ethereum-transaction-its-no-secret-its-just-smart`). Thus, the second transaction indicates that the owner called the "updateSeed" function. By decoding the input to the function (converting from the hexadecimal format to a normal string), it can be seen that the owner changed the value of the key to ";)".

How the transaction was executed:

1. Created a new Truffle project and copied "Assignment.sol" in the "contracts" folder.
2. Created a new Infura project in order to use its API to connect to a Ropsten endpoint.
3. Linked the Truffle project (using "truffle-config.js") to my Metamask wallet (using the "@truffle/hdwallet-provider" package) and my Infura Ropsten endpoint.
4. Then, the contract was compiled and Truffle's console was used to create an instance of the contract (using its deployment address).
5. This instance was used to execute the contract's "register" function in a transaction. The parameters used were k=";)" and UUN="s1519734". The transaction's hash is:

"0x84a83268c69d29c51db78261c5a3a5737902c987ca48ba2c2051be48c6f4fbad".

2 Part 2a: Token

2.1 High-level design decisions

2.1.1 Changing the token's price

This functionality is the most "thorny" aspect of the contract, so it's worth starting our discussion with it. It is perhaps worth to try and figure out what this contract can be used for, based on its API. We could find only two types of contracts in which changing the token's price is warranted.

The first type of contract corresponds to an Initial Coin Offering (ICO) contract, in which the token's price can change during the ICO, based on demand. However, an ICO has a limited supply of tokens, while our contract has an unlimited supply.

The other type is one mirroring "exchange" tokens such as "WBTC", where 1 WBTC token has a value equal to one bitcoin. To the best of our understanding, the price of this token fluctuates (artificially, by being changed by the contract owner) based on the exchange rate between ETH and BTC. However, people can trust WBTC because the people behind it have a reputation to maintain and the contract has gone through several audits (see: `https://blog.kyber.network/wbtc-progress-update-dec-2018-84e42b33843c`). Our contract will clearly not benefit from the same amount of trust.

The API of our token resembles the API of an ERC20 token, the major difference being that the token price can be changed by the contract owner. Presumably, our token could be used simply as a more convenient way to trade value, if its price is a subdivision of 1 ETH. However, if that's the case, then the owner having the capability to change the token price unchecked is a hazard to customers. After enough customers have bought tokens, the owner could change the price to zero, buy one token, set the token price to the contract's balance and then sell his token, thus emptying the contract's balance and stealing from all the users who bought the tokens.

This hazard is the reason why we have allowed only price increases. Naturally, by increasing the token's price, the contract's funds will no longer suffice to sell all the tokens. Therefore, the contract owner will need to provide enough ETH to the contract when calling the "changePrice" function, so that all the tokens can be sold for the new price.

Note: *An alternative would have been to restrict how much the token price is allowed to be decreased and also disallow a subsequent decrease for a period of time. This would make sense if there was a limited supply of tokens and if possessing those tokens provided some sort* *of real-world benefit to the users (e.g: access to a service provided by the contract owner). However this is not the case; the value of possessing our token is the same as its value in Ether. By decreasing its price, the contract essentially steals ether from its users, which would be unacceptable to them.*

Increasing/changing the token's price negatively impacts token transfers. If a price change occurs just before a transfer, then the user who initiated the transfer will end up over- or underpaying the recipient by $nr\_transferred\_tokens * |final\_token\_price - initial\_token\_price|$. One could argue that users would not be too happy if a price increase occurred just after they transferred some tokens either, although from our perspective it would be fair as users accept this risk by using the token. However, we consider that the first situation is not fair. We consider several mechanisms to mitigate this:

1. We could announce a price increase before it happens, using an event. However, this would lead to people buying tokens before the increase just so they will get free ETH (which is a hazard to the contract owner).
2. We could require the transfer initiators to be contracts and to have defined an "acceptTransfer" function, which returns a Boolean value indicating whether the current token price is the expected one. However, from the contract definition, it seems we cannot make this restriction.
3. Add a "desiredTokenPrice" argument to the "transfer" function. If the desired token price is not equal to the current token price (e.g: due to a price change), then the transfer would be reverted. This would be my preferred way of handling this issue, but to implement it we would change the API, which seems unacceptable for the assignment.
4. The fourth mechanism is the one we went with: we simply do not allow transfers immediately after a price change. In practice, we do not allow transfers occurring after fewer than 25 blocks (roughly 5 minutes in the main ETH net) after a price change. We appreciate that this is a major disruption to the customer, but it is preferable to him basically losing money. This also introduces the opportunity for the contract owner to grief customers with very small price increases, thus denying their ability to transfer tokens for extended periods of time. This is why the contract requires the price increases to be big enough. In the current implementation, the contract requires the new price to be at least double the old price. This is to both guard against this form of griefing and to disincentivise the owner from changing the price too frequently, since each price change causes some disruption to the token owners.

2.1.2 Global variables

The global variables are as follows:

- "address owner": a variable which holds the contract owner's address. Its value is initialised by the contract's constructor to the address which created the contract and cannot be changed once initialised. Only the owner can call the "changePrice" function.

- "uint256 transfer_timeout_period": How many blocks should be mined after a price change before transfers are allowed again. Set to 25 at contract creation, it cannot be changed.
- "uint256 last_price_change_block": Variable which keeps track of the block in which the last price change took place.
- "uint256 tokenPrice": Keeps track of the price of the token (in Wei). Initialised to 5e14 Wei in the constructor. This is so that after the required price doubling, its price will be 1/1000 ETH, which is approximately 0.7$. We want our token to be worth a useful subdivision of ETH.
- "uint256 nrSoldTokens": Variable which keeps track of how many tokens are currently in circulation. This variable is required by the "changePrice" function. Initialised to 1 due to gas fairness considerations (see Section 2.2)
- "mapping (address => uint256) owned_tokens": Dictionary which keeps track of the number of tokens owned by each address.

2.1.3 Buying tokens

Any user can buy tokens by calling the "buyToken" function and providing the "amount" of tokens they want to buy. Additionally, the function is payable and the value of the message must be equal to $amount * tokenPrice$. If this is satisfied, then the purchased amount of tokens is added to the sender's internal "wallet" (to owned_tokens[msg.sender]). Then, "amount" is added to "nrSoldTokens" and a Purchase event is emitted.

If a user tried to buy some tokens after a price increase, the function will likely fail as the message value will not be sufficient.

2.1.4 Selling tokens

A user can sell N tokens by calling "sellToken(N)". N tokens will be subtracted from his internal "wallet" (owned_tokens[msg.sender]). If the funds do not suffice, this will cause an underflow, which will be detected by the SafeMath library and the transaction will be reverted. If the funds suffice, N will also be subtracted from "nrSoldTokens". Then, the contract will transfer "N * tokenPrice" Wei to the user's wallet (msg.sender) and emit a Sell(msg.sender, N) event.

2.1.5 Transferring Tokens

As justified in Section 2.1.1, transfers cannot happen within "transfer_timeout_period" blocks from a price change.

The user who initiates the transfer is identified by "msg.sender", while the recipient and the amount transferred are arguments to the "transfer" function (as per the API). If the transfer is sufficiently spaced away from a price change, what happens next is that the number of tokens to be transferred are subtracted from owned_tokens[msg.sender] and added to owned_tokens[recipient]. As SafeMath is used for all arithmetic operations, if the sender's funds do not suffice, then owned_tokens[msg.sender] - amount will underflow, reverting the transaction.

The main issue of the ERC20 token standard is that users can use the "transfer" function to transfer tokens to contracts which have no capability of using them (either selling or transferring them). This essentially leads to the tokens and their value being forever lost. We chose to address this the same way as the author of the ERC223 token did (`https://github.com/Dexaran/ERC223-token-standard`). We detect whether the recipient address represents a contract; if it does, then the contract needs to implement "IERCiobReceiver" interface. The interface has a single function, "receiveERCiobTokens" which accepts the address of the sender and the transferred amount as arguments. What our "transfer" function attempts to do is to initialise an IERCiobReceiver object from the recipient's address and call its "receiveERCiobTokens" function. If this fails, then the transaction will be reverted. This ensures that the tokens will not be sent to a contract which lacks the capability of using them. A very simple implementation of the interface would be a contract that can save the address of the token contract and the amount of tokens and could then call its "sellToken" function and redirect the funds to its owner.

Finally, if nothing went wrong a "Transfer(msg.sender, recipient, amount)" event is emitted.

2.1.6 The "changePrice" function

Only the contract owner can call this function; if anyone else tries to, the transaction will be reverted.

The new price must be greater than the initial price; furthermore, to prevent the owner from griefing the users the new token price must be sufficiently greater than the initial price. In the current implementation, the new token price must be at least twice the original price, otherwise the transaction will be reverted.

Finally this function is payable and for the price change to go through, the contract owner must transfer sufficient ETH so that all the users can sell their tokens for the new price. In practice, if the new contract balance is not equal to nrSoldTokens * the new token price, then the price change will fail and the transaction will be reverted.

If the transaction is successful, then the "tokenPrice" and "last_price_change_block" variables are updated and a "Price" event is emitted with the new price.

2.1.7 Accessing token balance

A user can access his token balance by calling the ”getBalance()” function, from the account which was used to buy tokens. The function returns the amount of tokens owned by msg.sender (returns owned_tokens[msg.sender]).

2.1.8 Linking the ”SafeMath” Library

On line 3 in the code, we import the SafeMath library. As it contains only internal functions, when we deploy the contract (”ERCiobToken.sol”), the library will be embedded into the contract's bytecode by the EVM. This means that when we call a library function (such as ”add”) this corresponds to a ”JUMP” instruction in the EVM (same as calling one of the main contract's functions), which costs at most 10 gas.

If the library contained external functions, then it would have been deployed as a separate contract, and we would have needed to link this instance to our main contract (this is done automatically by Truffle when deploying the contracts; it would first deploy the library, then use its address to replace the library's reference in the main contract's bytecode when deploying the latter). In this case, whenever the main contract calls one of the library's functions this will instead correspond to a ”DELEGATECALL” instruction in the EVM, and would incur a much higher gas cost (700 according to `https://ethereum.github.io/yellowpaper/paper.pdf`, Appendix G).

2.2 Gas evaluation

2.2.1 Gas Costs

All the gas costs presented in this section have been estimated using Remix and Solidity version 0.7.4.

For each transaction, Remix indicates its **execution cost** and its **transaction cost**. The execution cost is the amount of gas needed to run the function called by the transaction, while the transaction cost = execution cost + gas price of sending the necessary data to the blockchain + 21k gas (fixed fee) - any refunded gas.

Two calls to the same function can incur different gas costs. The main reason for these discrepancies in our contract is variables' state change. In Solidity, changing a (storage) variable from zero to a non-zero value incurs a fixed 20K gas cost. Changing it from a non-zero value to another value incurs a 5K gas fee. ”Deleting” a variable (changing its non-zero value to zero) refunds 15k gas cost, with the special mention that the refund cannot exceed half the gas used for the current context. Source: `https://ethereum.github.io/yellowpaper/paper.pdf`.

buyToken
Case 1: User who calls the function owns no tokens.

Transaction cost: 51161 gas
Execution cost: 29697 gas

Case 2: User who calls the function already owns some tokens.
Transaction cost: 36161 gas
Execution cost: 14697 gas

The difference in gas cost comes from the fact that in the first case, updating owned_tokens[msg.sender] from 0 to "amount" incurs a 20k gas cost, while in the second case, since its value is now different from zero the operation will only cost 5k gas. This is why the execution of the function is 15k gas more expensive in the first case than in the second.

Case 3: The msg.sender does not provide sufficient funds (in which case the transaction is reverted).
Transaction cost: 22846
Execution cost: 1382.

transfer
Case 1: Recipient is not a contract; sender will have some tokens left after the transfer and recipient has no tokens before it.
Transaction cost: 54830
Execution cost: 31958

Case 2: Recipient is not a contract; sender sends all his remaining tokens, recipient already has some tokens.
Transaction cost: 24830
Execution cost: 16958

Case 3: Recipient is not a contract; sender will have some tokens left after the transfer, recipient already has some tokens.
Transaction cost: 39830
Execution cost: 16958

Case 4: Recipient is not a contract; sender sends all his remaining tokens, recipient had no tokens before the transfer.
Transaction cost: 39830
Execution cost: 31958.

Case 5: Recipient is a contract and it implements the required interface.
Transaction cost: 41670 (26670 if the contract already owned some tokens)

Execution cost: 33798 (18798 if the contract already owned some tokens)

Case 6: Recipient is a contract and it does not implement the required interface.
Transaction cost: 54948
Execution cost: 32076

Case 7: Transfer happened too soon after a price change (transaction is reverted).
Transaction cost: 25189
Execution cost: 2317

Case 8: The user who initiated the transfer doesn't own enough tokens (it is assumed that he does in all other cases).
Transaction cost: 26516
Execution cost: 3644

The difference in gas costs between cases 1, 2, 3 and 4 as well as between 5 and 6 can be fully explained by how the owned_tokens[msg.sender] and owned_tokens[recipient] variables change. Let instr1 = the instruction in which owned_tokens[msg.sender] changes and instr2 = the instruction in which owned_tokens[recipient] changes.
Case 1: instr1 costs 5000, instr2 costs 20000.
Case 2: instr1 costs 5000, but triggers a refund of 15000; instr2 costs 5000.
Case 3: instr1 costs 5000, instr2 costs 5000.
Case 4: instr1 costs 5000, but refunds 15000; instr 2 costs 20000.
instr1 and instr2 are the only two instructions whose gas cost differ between those four cases. Analogous discussion for case 5.

The execution costs for case 5 are slightly higher since the two instructions inside the "if" statement get executed.

I do not consider that these variations in gas costs are avoidable. If the sole purpose of a user is to buy and transfer some tokens (similar to paying for a service), it would be advisable for him to buy exactly the number of tokens he needs to transfer in order to minimise the used gas.

sellToken
Transaction cost: 43872, if the user does not sell all his tokens, 28872 otherwise
Execution cost: 22408
The difference in the transaction gas costs is due to how the "owned_tokens[msg.sender]" variable changes in the two cases (if the user sells all his tokens, this variable becomes zero, refunding 15k gas).

If the user owns fewer tokens than how many he wants to sell, the transaction cost will be 23188 gas (with a 1724 execution gas cost).

changePrice
The costs for the first call:
Transaction cost: 58730
Execution cost: 36946
The costs for every subsequent call will be 15k gas cheaper (due to last_price_change_block being changed from a non-zero value to another non-zero value as opposed to the first call, where it is changed from zero to a non-zero value).

getBalance
This function does not modify the state of the contract or the blockchain, so it can be directly called (0 gas cost).

Deploying the contract
Transaction cost: 1110571 (without compiler optimisation), 870929 (with optimisation enabled, optimizer-runs=10000)
Execution cost: 816135 (without optimisation), 640901 (with optimisation).
The cost of deploying the contract is directly proportional to the size of its bytecode after compilation. Therefore it is in our interest to keep the contract as small as possible.

2.2.2 Optimising the gas costs

Making the contract functions external
An "external" function in Solidity can only be called from outside the contract. Additionally, for external functions, the arguments can be read directly from calldata whereas for public functions Solidity automatically saves the arguments in memory (as per `https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices`), which will incur additional gas costs. All of the functions of the contract are meant to be called from outside the contract anyway, so making them "external" requires no changes to our code and saves gas.

SafeMath
As explained in Section 2.1.8 by embedding SafeMath into the contract's bytecode instead of deploying it as a separate contract, we save at least 700 gas per SafeMath function call. Each function except getBalance has 3 such calls, so we save at least 2100 gas per function call.

Fairness - nrSoldTokens
If we inspect the code, we can see that we initialise "nrSoldTokens" to 1 in the contract constructor. This is because we consider that the owner should burden the initial gas cost of incrementing this variable from zero and this change allows for the gas costs of "buyToken"

and "sellToken" to be symmetric for all token users. The calculations using this variable are modified accordingly.

Storage variables
As we saw at the beginning of this section, working with storage variables is generally expensive; this is why we should be certain that the ones we use are actually needed and that we are using them as efficiently as possible.

A potential optimisation we could make is to use different types for "transfer_timeout_period", "tokenPrice" and "nrSoldTokens", since uint256 might be too large. "transfer_timeout_period" should not be greater than 2^6, while the tokenPrice should probably not be higher than 10^3 ETH = 2^{70} Wei, so it could fit in a uint128. Similarly, "nrSoldTokens" is bounded by the current available ETH divided by the tokenPrice (in ETH). For a tokenPrice of 10^{-3} ETH and a supply of 2^{26} ETH, "nrSoldTokens" will not be greater than 2^{36}. All in all, these three variables could fit into a single uint256, saving storage space and thus gas; however, this comes at the cost of upper bounding the number of tokens that can be owned at any time (+ lower and upper bounding the token price), which we consider to be counter to what is requested in the assignment (+ it is specifically requested that tokenPrice should be a uint256).

Using memory variables
Initialising and modifying memory variables is much more cheaper than working with storage variables. This is why they should be used whenever possible.

Contract size
The gas cost of a contract's deployment is directly proportional with the size of its bytecode (and thus with the size of the contract). We made sure that every instruction and variable declaration is needed. To reduce the deployment cost even further, we could remove SafeMath's "div" and "mod" functions, as our contract does not use them.

Compiler optimisation
We compiled the contract with optimisation enabled and optimizer-runs set to 10000. This led to the contract deployment being significantly cheaper (by 250k gas) and for each function call to be marginally cheaper (by 200-400 gas, not reflected in the values presented in this report - except for "changePrice").

2.3 Hazards and vulnerabilities

2.3.1 Overflows and underflows

All our numeric variables are uint256. The normal arithmetic operations involving uint256 were replaced with SafeMath's ("+" was replaced by "add", "-" by "sub" and "*" by "mul"; no other operation is used in our contract), which revert the transaction if they caused an underflow or an overflow.

Even though our contract is impervious to these errors, we do not want them to happen often (if at all). It is perhaps important to consider what would happen if the tokenPrice is too low. The number of tokens that can be sold in the main ETH net (if everybody were to buy tokens with their ETH, which is practically impossible) is

$$max\_nrSoldTokens = \frac{current\_ETH\_supply}{min\_tokenPrice}$$

. The ETH supply at the beginning of 2020 is estimated to have been around 2^{26}. The smallest tokenPrice possible in our implementation is 1 Wei = 10^{-18} ETH. Thus $max\_nrSoldTokens$ on the main net will be $10^{18} * 2^{26} = 2^{86}$ ETH. Therefore, we would need at most 86 bits to represent "nrSoldTokens" in the worst case scenario. As this variable is saved in a uint256, it overflowing (or a user's internal "owned_tokens" entry) is currently practically impossible.

2.3.2 Griefing the users

As mentioned previously, the capability of changing the token price by the contract owner is not reasonable for users without the contract (and backend) having been audited by at least one reputable company or without the owner being well-known and having a good reputation himself. Additionally, changing the token price impacts every function of the contract, with transfers being impacted the most. Therefore, to make this token usable we need to restrict the contract owner's power of changing the token price.

If the contract owner is free to decrease the token price as he wants, he could make it zero, buy one token, make the token price equal to the contract balance and sell it, essentially stealing all funds from every legitimate user. Any security conscious user would not use the contract in this case. Therefore, we need to either restrict how much the token price can be decreased in a time interval or forbid the owner from ever decreasing it altogether. The first option would make sense if the value of the token would be tied to something external, such as tokens which mirror other currencies (e.g: WBTC, where 1 token = 1 Bitcoin, or a token which would be equal to 10 dollars). Even so, how the token price changes would be left completely to the latitude of the owner, so an external audit might be warranted to ensure fairness. This is not a possibility in our case, so we decided to forbid the owner from decreasing the token price.

Changing the token price might cause "buyToken" operations to fail, but this does not necessarily lead to any losses on the behalf of the customers - they can quickly retry the operation with a corrected amount of funds. "sellToken" operations are affected in the sense that a user might end up selling more tokens that he would have otherwise wanted. The user can always buy some tokens back, but he will lose the gas required to do this operation. Transfers are most affected by token price increases, as a user might end up transferring more tokens than he wanted. This is why transfers within 25 blocks (roughly 5 minutes) of a price change are forbidden.

However, this leads to another hazard: the owner can continuously prevent transfers by increasing the token prices by very small amounts, leading to potential customer losses (e.g to customers involved in time-gated transactions). This is why we want to ensure that the new token price is sufficiently larger than the old price (in the current implementation, the new price should be at least twice the old one). We wanted the token price to be exponential in the number of price increases, as a disincentive for the owner to do consecutive price changes (as he needs to provide the additional funds necessary to sell all the currently owned tokens when increasing the price).

2.3.3 Re-entrancy attacks

As the API did not allow for a pull over push pattern - aka the "sellToken" operation should directly refund the ETH to the customer - we preferred to use the "checks-effects-interactions" pattern in all the functions of our contract to guard against re-entrancy attacks.

For example, we could analyse "sellToken", with which a user wants to sell "x" of his tokens.
Checks
The checks are performed by the SafeMath library, when we subtract "x" from the sender's balance, owned_tokens[msg.sender] (and from "nrSoldTokens"). If the sender does not have sufficient tokens, this operation will underflow and SafeMath will revert the transaction.
Effects
"x" tokens are subtracted from the sender's balance and from "nrSoldTokens".
Interactions
"x" * "tokenPrice" Wei are transferred to the sender's address and a "Sell" event is emitted.

A double-spend re-entrancy attack will thus not be possible.

2.3.4 Forcibly sending ether to the contract

ETH could be sent to the contract's address before its creation. This could lead to the owner having some difficulty in changing the token's price. In the previous implementation, we required the contract's new balance after the price increase to be exactly equal to the new price times the number of sold tokens. If an attacker provided sufficient funds to the contract, price changes could have been impossible before sufficient tokens had been sold (or without the owner increasing the price more than he'd wanted).

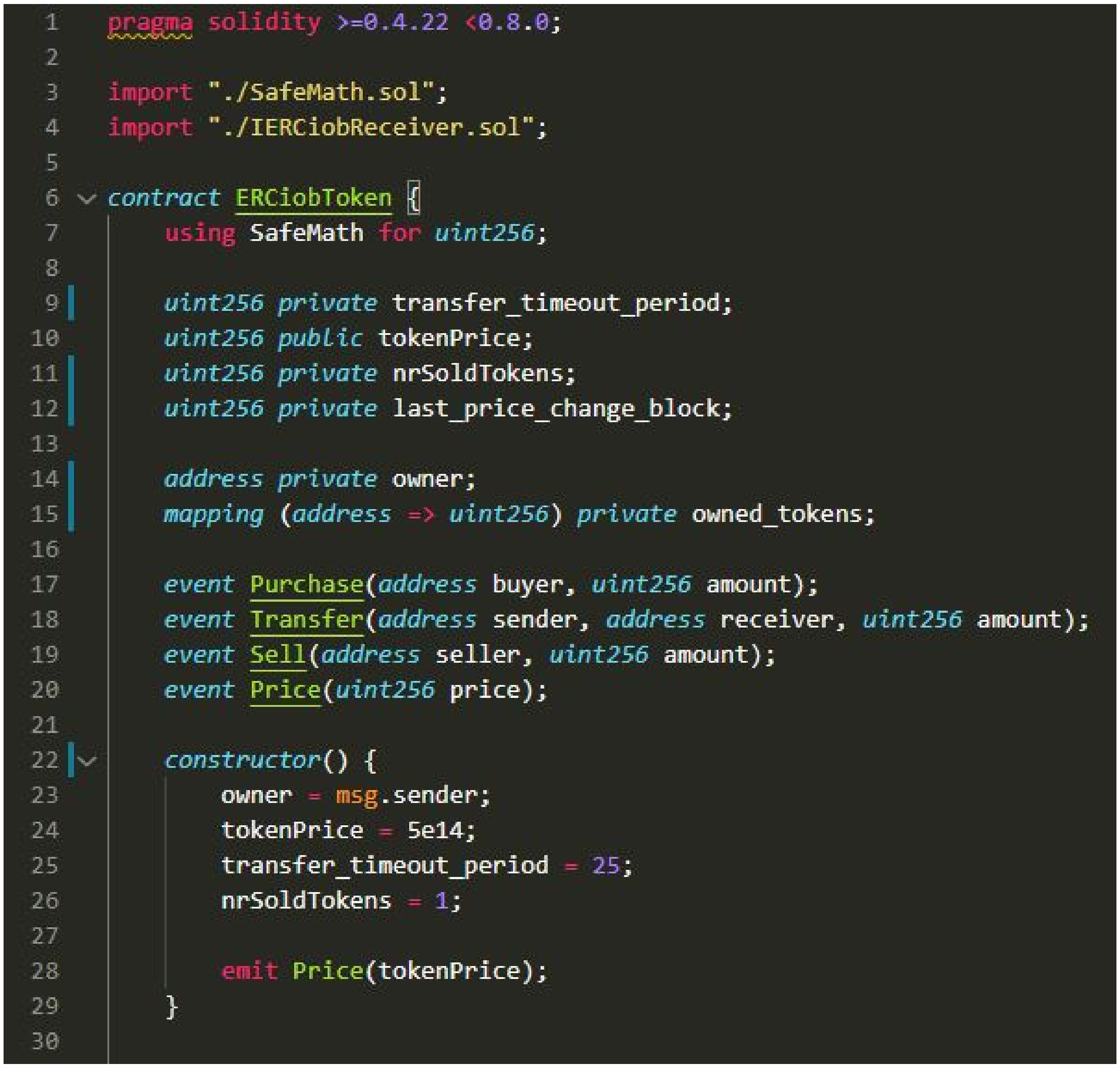
The fix to this is simple: we will require the contract's balance to be greater than the new price times the number of sold tokens and we will refund the difference to the contract owner.

2.4 Transaction histories

My contract deployment transaction:
"0x3a81be9787ca96b9fc0e99b7b2657a5020ec08d7c5b71059d4543c75edf8ecea"
My contract's address:
"0x4c9Bf1d037Ea33E550438523A0636c59bff7743C"
First token bought on my contract (tx hash):
0x705107859a30598d95fa53052f88a89f27180984803a2a0e94373065c15c531d
Price doubling (tx hash):
0x424c7628b9f523eccaee812f8f49bc30d77e6c527598c060d61c72b7372807bd
Partner's (s2125644) contract address:
0xcc221c2fc2d08cddee81b10c464c12fe47237bfb
Tx in which I bought two tokens:
0xc416ad8927cb3ae693c65a9dad14799fba74222fe77c493238b018a0828c336e
Tx in which I transferred 1 token to the contract owner:
0x32eca5a619b4635c2096e633e5d6cc8c067d611d20ac9a03a3e6e47b3e11370e
Tx in which I sold the remaining token:
0xc9b08f67380ca7815b396f4039be1956024a15a39644f6a65abb8f8497f8c265

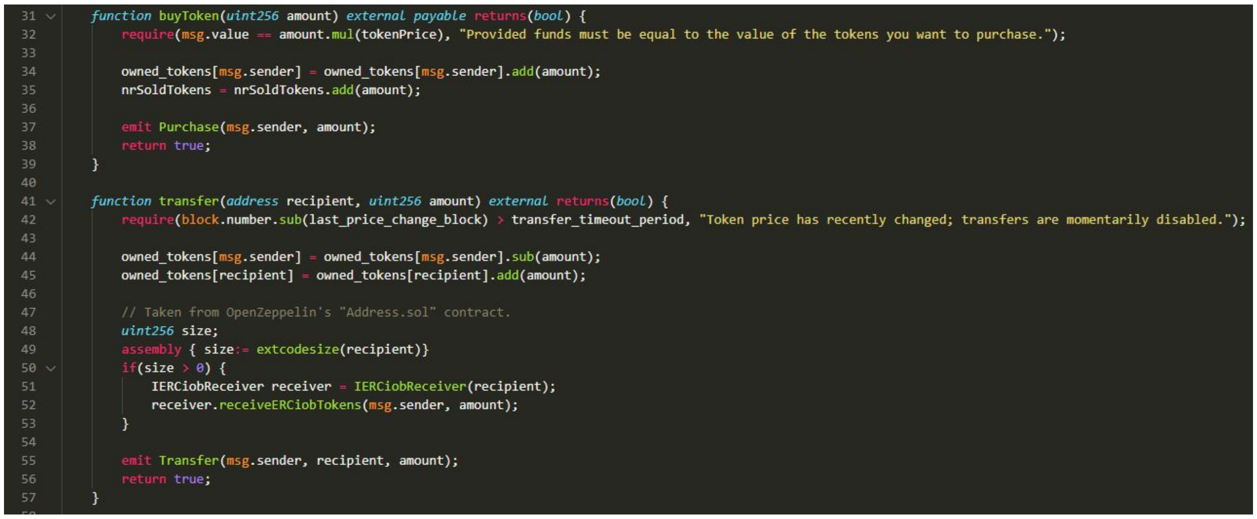
2.5 Contract code

The contract was compiled with Solidity v0.5.16.



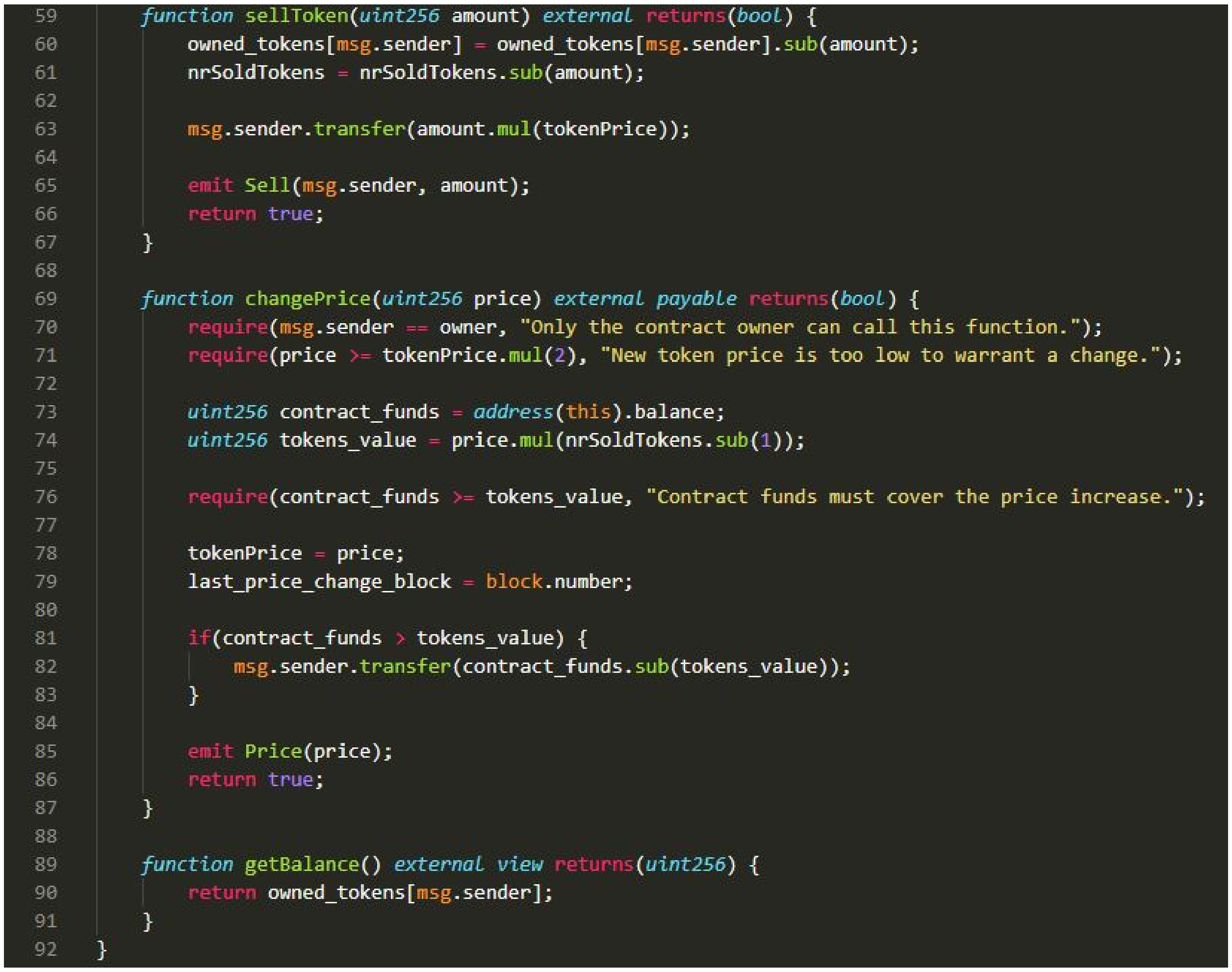
```
pragma solidity >=0.4.22 <0.8.0;

import "./SafeMath.sol";
import "./IERCiobReceiver.sol";

contract ERCiobToken {
    using SafeMath for uint256;

    uint256 private transfer_timeout_period;
    uint256 public tokenPrice;
    uint256 private nrSoldTokens;
    uint256 private last_price_change_block;

    address private owner;
    mapping (address => uint256) private owned_tokens;

    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() {
        owner = msg.sender;
        tokenPrice = 5e14;
        transfer_timeout_period = 25;
        nrSoldTokens = 1;

        emit Price(tokenPrice);
    }

```

Figure 1: Contract constructor and storage variables.



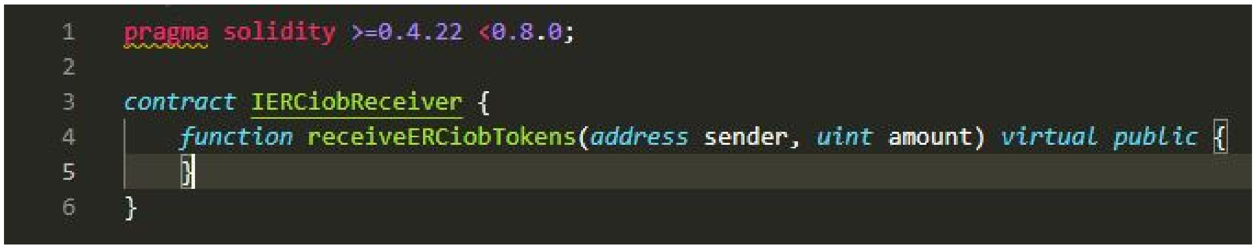
```
    function buyToken(uint256 amount) external payable returns(bool) {
        require(msg.value == amount.mul(tokenPrice), "Provided funds must be equal to the value of the tokens you want to purchase.");

        owned_tokens[msg.sender] = owned_tokens[msg.sender].add(amount);
        nrSoldTokens = nrSoldTokens.add(amount);

        emit Purchase(msg.sender, amount);
        return true;
    }

    function transfer(address recipient, uint256 amount) external returns(bool) {
        require(block.number.sub(last_price_change_block) > transfer_timeout_period, "Token price has recently changed; transfers are momentarily disabled.");

        owned_tokens[msg.sender] = owned_tokens[msg.sender].sub(amount);
        owned_tokens[recipient] = owned_tokens[recipient].add(amount);

        // Taken from OpenZeppelin's "Address.sol" contract.
        uint256 size;
        assembly { size:= extcodesize(recipient)}
        if(size > 0) {
            IERCiobReceiver receiver = IERCiobReceiver(recipient);
            receiver.receiveERCiobTokens(msg.sender, amount);
        }

        emit Transfer(msg.sender, recipient, amount);
        return true;
    }
```

Figure 2: ”buyToken” and ”transfer” functions.



```
    function sellToken(uint256 amount) external returns(bool) {
        owned_tokens[msg.sender] = owned_tokens[msg.sender].sub(amount);
        nrSoldTokens = nrSoldTokens.sub(amount);

        msg.sender.transfer(amount.mul(tokenPrice));

        emit Sell(msg.sender, amount);
        return true;
    }

    function changePrice(uint256 price) external payable returns(bool) {
        require(msg.sender == owner, "Only the contract owner can call this function.");
        require(price >= tokenPrice.mul(2), "New token price is too low to warrant a change.");

        uint256 contract_funds = address(this).balance;
        uint256 tokens_value = price.mul(nrSoldTokens.sub(1));

        require(contract_funds >= tokens_value, "Contract funds must cover the price increase.");

        tokenPrice = price;
        last_price_change_block = block.number;

        if(contract_funds > tokens_value) {
            msg.sender.transfer(contract_funds.sub(tokens_value));
        }

        emit Price(price);
        return true;
    }

    function getBalance() external view returns(uint256) {
        return owned_tokens[msg.sender];
    }
}
```

Figure 3: ”sellToken”, ”changePrice” and ”getBalance” functions.



```
pragma solidity >=0.4.22 <0.8.0;

contract IERCiobReceiver {
    function receiveERCiobTokens(address sender, uint amount) virtual public {
    }
}
```

Figure 4: Token receiver interface.

3 Part 2b: KYC Considerations and Token Issuance

3.1 Additional constraints

The solution must respect the following constraints:

1. ”The only two parties that should have access to the identification document should be the identity owner (who creates it) and the smart contract owner.” (Piazza @215)
2. ”In the context of the assignment, you should assume that the only channel of communication between the buyer and the contract’s issuer is the blockchain.” (Piazza @219)
3. ”You should assume that the token issuance happens only if the provided identification document is correct.” (Piazza @223)

3.2 Assumptions

1. Constraint #1 led us to exclude any solution which would use an on-chain KYC provider (aka we cannot delegate the KYC process to somebody else or to another company/government). If this were the case, the third party could have blindly signed a trusted user’s message to our contract. This also excludes solutions using digital certificates. Therefore, we need to do the KYC process ourselves.
2. As a consequence of constraint #2, we assume that a user can satisfactorily prove his identity to us in a single string message. This is unrealistic in a real-world use case, as most KYC solutions require a user to provide proof such as pictures of their government issued ID and images or videos of themselves (and perhaps documents such as bills, crime record etc.). If we relaxed this constraint, the user might be able to send us a link to an encrypted folder containing all this data, which can be unlocked (say, using the user’s account address).
3. Furthermore, we assume that the text used to identify a user uses only ASCII characters and is smaller than 450 characters. This is reasonable if the user sends us a URL (or a shortened URL).
4. It is assumed that all the users have access to the contract’s code and storage variables (the owner can simply make the code public on a service such as Etherscan).

3.3 Solution for identifying a user

Given the stated constraints and assumptions, the proposed solution is:

- The owner of the contract creates an RSA 2048-bit key-pair, and saves the public key in the contract. The maximum size of the data that can be encrypted with the public key is 245 bytes == 490 ASCII characters (assuming an 11-byte overhead due to padding = we are using PKCS#1 v1.5).
- The user appends the address of his account (which is 40 characters long) to his link/identifying message (which needs to be under 450 characters long). The resulting string is encrypted using the contract's public key. This is done off-chain since Solidity does not seem to have this capability.
- The user then sends this encrypted message/string to the contract.
- The contract emits an event containing the message and the sender's address.
- The contract owner monitors events emitted by the contract (perhaps through an automated script) and once he sees that a user wants to register, he then decrypts the string using his private key, obtaining an address and the identifying message. If the decrypted address matches the one which sent the encrypted string, the KYC process can start (which is entirely off-chain). If the addresses do not match then the request is ignored and encrypted string is deleted from the contract's memory.
- If the KYC process is successful, then the contract owner will add the user's address to the list/dictionary of verified addresses, and the now verified user can start buying and trading tokens.
- Verified addresses which have been inactive for a long period of time will be removed from the list/dictionary, since storage is expensive. The monitoring (and the triggering of the account deletion) will be done off-chain by a script.

3.4 Necessary API modifications

Additional storage variables:

- "mapping (address => bool) verified": A dictionary, which indicates whether an address has been successfully verified or not.
- "string publicKey": The key the user will use to encrypt his address and identifying message.

Additional events:

- "event RequestID(address user, string message)": Event which is emitted after a user requests to be registered.
- "event Register(address user)": Event which is emitted after a user has been registered.
- "event Delete(address user)": Event which is emitted after a user's address has been deleted for inactivity.

Additional functions:

- "beginRegistration(string message)": Function which a user calls in order to begin the identification/registration process. Must provide the encrypted identifying string + the user's address.
- "register(address user)": Registers the user's address (after a successful identification process). Only callable by the owner.
- "delete(address user)": Un-registers a user's address (after a period of inactivity).

Modified functions:

- "buyToken": Now requires the message sender to be verified (verified[msg.sender] == true).
- "transfer": From the assignment description, tokens can only be issued to (created by) verified users. It is unclear whether a verified user should be allowed to transfer tokens to an unverified one. My assumption is that this should not be allowed, in which case a transfer will now fail if the recipient is unverified.

3.5 Is it possible to implement this process entirely on-chain?

My answer is that this does not currently seem possible. The key issue is that there does not seem to be a way to associate an identity with an address in a verifiable manner using only on-chain data, while preserving a user's anonymity. Most of the KYC solutions on the ETH blockchain rely on an on-chain authority/contract which attests whether an address belongs to a verified user or not and on an off-chain identification process (e.g: `https://www.serto.id/`, with its "ethr-did-registry").