Assignment 3

Pablo Miró - s1814033
BDL - Blockchain and Distributed Ledgers

January 24, 2022

Contents

1 Interacting with smart contract in Ropsten

1.1 Obtaining Ropsten Ether

The first step to interact with the smart contract given is to obtain some Ropsten Ether, since it has been deployed in Ropsten, Ethereum's testnet. I tried connecting to Metamask's faucet but, unfortunately, it seemed that it did not work and was under maintenance as far as I found online. Consequently, I ended up getting Ropster Ether from a different faucet.

In order to get some Ropsten Ether to my address, I just copied the Ethereum address I use for the BDL course and submitted it in the faucet so that I would be given 0.3 `ETH`.

1.2 How to obtain key to register

Looking at the code for the smart contract we find two functions alongside one constructor. We must understand the code before we can see how to find the key the contract is using.

To begin with, we know that the constructor is run when the smart contract is deployed:

- It has an argument `k` which will be given by the owner of the contract, and it will be the initial key.
- The function `updateKey(string memory k)` lets the owner of the contract change the key that the smart contract is using.
- Lastly, the `register(string memory k, string memory uun)` function takes two parameters: a key and a unn. If the key given to the function is correct, the unn will be added to the `students[]` mapping in the contract.

To obtain the required key for registering in the contract we can look at the blockchain for: the transactions the owner made, the original key was when the contract was first deployed in Ropsten, if the key was ever updated and what was it changed to.

To begin with, we go to Ropsten Testnet explorer and we look for the owner's address, i.e. `0xC8e8aDd5C59Df1B0b2F2386A4c4119aA1021e2Ff` (we can obtain this address by calling the public `address` variable `owner` in the smart contract).

Looking at the owner's history of transactions we can see that the last two, at the time of this writing, are related to the smart contract `0xde3a17573b0128da962698917b17079f2aabebea`: The owner deployed the contract to Ropsten, and they updated the key. If we go to 'search more details' for the last transaction we can scroll to the end and check what the input data was. If we press 'decode input data' we will see the string '`actually...;)`', which corresponds to the argument given to the function `updateKey(string memory k)`. This transaction corresponds to the last time the owner called `updateKey(string memory k)`, hence we obtained the key!

N.B. We can also check what the key first key of the smart contract was, i.e. the argument given to the constructor in the deployment of the smart contract. By looking at the transaction for the deployment of the contract (link), we can also obtain the input data and search for the constructor's argument. The input data is a very long string, but the argument of the constructor is appended to the end. Thus, the argument in binary corresponding to the original key is `146e6f70652d746869732d69732d7468652d6b6579` which, translated to ASCII means `nope-this-is-the-key`.

1.3 Registering the UNN in the smart contract

Having obtained the current key stored in the smart contract (1.2) we can add our student `UNN` to the contract's `string[] students` using the `register` function.

I called the function with the arguments key = actually...;) and unn = s1814033. It can be checked that I am indeed registered in the contract by looking at the value of `students[14]`, which is my student UNN.

2 Custom Token

2.1 High-level decision making

2.1.1 Internal variables

My ‘CustomToken‘ smart contract has 4 state variables, with 3 of them being `internal` and the other one being `public`. The `internal` variables are:

- `address internal owner`. This state variable keeps the address which deployed the contract (which I will consider the *owner* of the smart contract). It will be used in `changePrice(...)` to check the caller of the function is the address who deployed the contract.
- `mapping(address => uint) internal balances`. Solidity mapping which stores the balances of those addresses that have used the contract to buy *bdlToken* tokens.
- `uint internal totalToken`. This variable keeps track of the circulating supply of BdlToken. We need this variable to check whether the owner can change the price of the contract. More on this in 2.1.2.

2.1.2 Buying/selling tokens

In order for the customer to **buy tokens** they need to use the `buyToken(uint amount)` function. This will obviously require that the customer has sent enough funds to pay for the number of tokens s/he wants to buy. This amount is calculated by multiplying the number of tokens times the price of each token. If the sender sends the exact amount or more, the contract will update the balance of the sender (note that if the caller sends more funds those will be kept by the contract). Otherwise the transaction will be reverted. The `buyToken` function updates the customer balance and the tokens in supply apart form emitting a `Purchase` event and returning `true` when successful.

When a customer wants to **sell tokens** they own, they will call the `sellToken(uint amount)` function. The contract will carry out some checks: To begin with, the customer should have sent a minimum of `1 wei` since the `customSend` function used by the contract sends `1 wei` to the owner of the `CustomLib` library and we want the user to receive `tokenPrice` wei per token sold. The contract will also check that the customer has previously bought the number of tokens they are planning on buying. Lastly, it will check whether the contract has enough balance to buy the tokens the sender wants to sell. The `sellToken` function keeps track of the customer balance by subtracting the tokens they sold, it will also update the # of tokens in circulating supply as well as emitting a `Sell` event and returning `true` if the transaction was successful.

2.1.3 Changing the price of the tokens

This can be done through the `changePrice(uint price)` function. Not everyone will be able to change the price of the BdlToken, but only the owner of the contract. This is expressed by the `require(msg.sender == owner)` statement in the code. After this, the contract checks whether it has enough funds to buy all the tokens in circulating supply in case every user who owns BdlToken tokens wants to sell all of their funds. If the contract has enough funds, it will change `tokenPrice`, emit a `Price` event and return `true`. Otherwise the transaction will be reverted.

For the owner to be able to increase the price of BdlToken we would need the contract to have received more funds than those obtained from customers buying BdlToken tokens. In order not to change the API given in the coursework file, I decided that this could be done by sending more money through `buyToken`. That is, we can call `buyToken` with `amount = 0`, but sending wei with the transaction, which will be kept by the contract.

Obviously in real life the price of a token changes according to supply and demand in exchanges, but since we have a dummy smart contract trying to replicate a token, and we need a function to change the price, that is how I thought it could be done.

2.1.4 Process of transfering tokens

According to the API of the contract, any owner of BdlToken will be able to send funds to another address, even if that address has not interacted with the contract. This is done via the `transfer(address recipient, uint amount)` function which will check that the sender has at least the tokens they want to send. If they do, it will subtract those tokens from the sender and add them to the recipient's account. Moreover, it will emit a `Transfer` event and return `true`. Otherwise, the transaction will be reverted.

2.1.5 How users access their token balance

Any customer will be able to call the `getBalance()` function to check how many tokens they possess. The contract keeps track of each user's balance with the `balances mapping` that we mentioned in section 2.1.1. Thus, the contract will check the number of tokens (`uint value`) that corresponds to the user (`address key`).

2.1.6 How the library is linked to the contract

We can have two possible scenarios depending on the type of the library we want to link:

1. We have an **embedded library**. This is the case if our library only contains internal functions so that it could be appended to the end of our smart contract. This library does not need to be deployed for the contract to use, and is usually kept in a separate file to improve readability of the contract.

2. We have a **linked library**. This happens when it contains `public` and/or `external` functions. In this case we do need to deploy the library, which will generate an address in the blockchain. This address will have to be used to link the library with our contract.

The provided `customLib` library contains a `public` function and it has been deployed to the blockchain. Hence, this library needs to be linked to our BdlToken smart contract.

In order to link the library, we will first need to use its functions in our code: We import the library by using an `import` statement and tell Solidity that that we will use `customLib`'s functions with `uint` and `address` variables as follows:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;
import './CustomLib.sol';

/** @title Custom Token contract. */
contract BdlToken {
    using CustomLib for uint;
    using CustomLib for address;
    ...
}
```

I used `'./CustomLib.sol'` because I have the code of the library in the same directory. Moreover, I modified the code in `sellToken` so that the contract uses the `customSend(uint256 value, address receiver)` function from `customLib`. The library functions have to be called as attribute to the first argument of the function. For example, if we want to call `customSend` with `uint amountToPay` and `address receiverAddress` we will need to write `amountToPay.customSend(receiverAddress)`.

Now, to link the already deployed library to our smart contract we need to change the **artifact** of the contract (a JSON file corresponding to the contract). If using remix we can make it generate metadata for the smart contracts, i.e. their **artifacts**, by activating the tab *General settings → "Generate contract metadata ..."*.

Having done so (and after modifying the contract to use the library's functions), I compiled the contract and modified its **artifact**, `BdlToken.json`. I looked for those objects which contain a `CustomLib` field with the value `"<address>"` and replaced that value with the address of the deployed library (given in the coursework document). Then, I also modified the field `autoDeployLib` to `false` so that it does not change the address.

This way, the smart contract will be using the `CustomLib` that was already deployed on Ropsten at address `0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47`. The beginning of the `BdlToken.json` file would look like this:

```json
{
  "deploy": {
    "VM:-": {
      "linkReferences": {
        "contracts/CustomLib.sol": {
          "CustomLib": "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
        }
      },
      "autoDeployLib": false
    },
    "main:1": {
      "linkReferences": {
        "contracts/CustomLib.sol": {
          "CustomLib": "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
        }
      },
      "autoDeployLib": false
    },
    ...
  }
}
```

2.2 Gas evaluation

In this section I will discuss how much gas the smart contract consumes, how it could be improved and how the linked library affects this cost.

2.2.1 Cost of deploying and interacting with the contract

Transaction cost is in units of gas (`gwei`).

Deployment of the contract

- Transaction hash: 0x4f5e44d56a978b0c0598d41674eaa98524ed085b765370cd14169e14b9291664
- Transaction cost: 662,728 gas
- Address of smart contract: 0x8310BCA669C84F98BaE0C19eF8a9047568CE0227

Calling the `buyToken` function

When a given user has not bought any tokens, its corresponding value in the mapping is initialized to 0. The first call to `buyToken` from that user will be more expensive (70,481 gas) while the subsequent ones will be cheaper (36,281 gas). Buying 1 token for the first time:

- Transaction hash: 0x3e5918e07243f7a3643ab1d23b1874c200542ec9b14506d6d0d3a77f84fee7e7
- Transaction cost: 70,481 gas

Buying 10 tokens after having bought 1 token:

- Transaction hash: 0x33cb4c93914241fafbda536e10cda4790074274a3173b3dda06db7941c3c5a5f
- Transaction cost: 36,281 gas

Calling the `transfer` function

Using the `transfer` function to send 1 token to a random address (I chose to send it to `address(0)`).

- Transaction hash: 0xb9ce674a58ee401cd4836608f336e279ffb8721d997b571d8614523f04355639
- Transaction cost: 52,180 gas

Using `sellToken`

Calling the `sellToken` function to sell 1 BdlToken. I sent 1 wei with the transaction because the `customSend` function from `customLib` sends 1 wei to the owner of such library. It would be possible not to send any wei in the transaction (remove my `require(msg.value >= 1);` statement) and change the code so that the user receives the price of the tokens minus 1 wei, but since the coursework states that the user "...receives from the contract tokenPrice wei for each sold token" I did it this way. The user will receive all the money that corresponds, but will have to pay for the 1 wei that "costs" to use `customSend`.

- Transaction hash: 0x19dc1193a6db707c403096d9330cc5968087c65f6fac6bf20f33c3da1968711b
- Transaction cost: 57,695 gas

Using `changePrice` function

Transaction for changing the price of the token from `1000 wei` to `900 wei`:

- Transaction hash: 0x76735f46e66345e84e13def24d061bb8c1a5e31a24139c2afe9cc64b79426ae1
- Transaction cost: 32,597 gas

The cost of `changePrice` remains constant no matter what the value of the parameter `uint price` is. I tried with bigger and smaller values of `tokenPrice` (obviously making sure the contract had enough funds so that I, the owner, could increase the price) and the transaction cost was the same.

Using `getBalance` function

getBalance is a `view` function and calling it externally does not cost any gas. It returns the number of BdlToken tokens the caller of the function has.

Getter for `tokenPrice`

Solidity provides automatic `getters` for storage variables declared `public`, as `tokenPrice`. It does not cost gas since we are reading the variable externally.

2.2.2 Techniques to make the contract more cost effective

There are many things one can consider when trying to make the contract cost less gas to interact with. I will outline the different things I considered to improve gas efficiency:

Constructor

The first thing I considered was the need for a constructor. In the beginning, the smart contract I wrote had the following lines of code:

```
address internal owner;

constructor() {
    uint tokenPrice = 1000 wei;
    owner = msg.sender;
}
```

Nevertheless, since those variables are initialised to values we know beforehand (the `owner` is the address with which I deploy the contract and the `tokenPrice` is set by the tokenomics of the team before deploying the contract [in this case it was set by me]). Therefore, by removing the constructor I expected to save `gas` and make the contract more efficient. I tried this out in code and found out that I only saved `44 gas`, which is not much, but is still an improvement. Then, I found out that if I initialised and declared the variables in the constructor, instead of declaring it above it, the cost of the deployment is the same. Consequently, I changed the code to declare and initialise the variables at the same time.

State variables

Another way of reducing the cost of the contract is related to its **state variables**. In `Ethereum` the state variables are stored in `storage` so they remain permanently in the blockchain. If we have variables which we do not need between function calls it is cheaper to use `memory`, where we hold temporary values. Hence, I tried to reduce the number of those state variables in the contract: I have `tokenPrice`, which is part of the API, and 3 other variables needed to store the balances, the address of the owner and the circulating supply of tokens.

Modifiers

Having reduced the number of state variables, I looked at the effect of the `modifiers` in Solidity. I read in the Solidity documentation that the `public` modifier can be more expensive because in `public` functions Solidity immediately copies array arguments to `memory`, while `external` functions can read directly from `calldata`. Nevertheless, I did not find a reduction in gas cost when I called the functions after changing the modifier to `external`. This is because `external` functions are not always more efficient, but usually when using **large arrays of data**.

Even though I believe keeping the functions as `external` is better practice since we do not call them internally in the code, I left the `public` modifier the coursework document specifies that the smart contract should follow the given **public API**; hence I left the functions as `public`.

On the other hand, by setting the state variables as `internal` instead of `public` (I left `tokenPrice` as `public` since the coursework mentions the API is `public`), I saved `72,069 gas` when deploying the smart contract (it costs 662,728 instead of 734,797 gas), which is a noticeable amount.

Data types

In the contract I use `uint` variables (shorthand for `uint256`) because of the way in which the Ethereum Virtual Machine works: Each storage slot has 256 bits. Storing, say, a `uint8` will cost more gas because the `EVM` will fill up all the missing digits with zeros. Moreover, `EVM` performs calculations in `uint256` so any type other than `uint256` will have to be converted as well.

I used a `mapping` for storing the users' balances, instead of an `array`. Due to the way the `EVM` works, `mappings` are usually cheaper than `arrays`: an `array` is not stored sequentially in memory but as a `mapping`. There are some situations in which we would prefer to use `arrays` e.g. when we want to iterate over some values or we want to pack smaller elements. Nevertheless, it is cheaper to access the balances of our contract using a `mapping` because it acts like a hash table, while in an `array` we would need to loop over all values.

2.2.3 Gas impact of the deployed library instance

Removing the call to `customSend` and using `payable(msg.sender).transfer(amountToPay)` instead results in the deployment of the smart contract to be more expensive, namelly it costs 612,667 gas instead of 662,728 ($\sim 50,000$ gas units cheaper).

Furthermore, the transaction cost for selling a token without using `customLib`'s function drops from 57,695 to 34,904 gas. Therefore, it is clear that using the deployed library is more expensive than using the `transfer` function. This makes sense because when calling `sellToken` we have to call to another `public` function which increments the cost of the transaction.

2.3 Potential Hazards and vulnerabilities

There are several ways the contract can be vulnerable. In this section I will explain how the contract could be found vulnerable and how I avoided them.

2.3.1 Griefing

To understand what a griefing attack is, it is important to know that a smart contract will call its default function when it receives money without any extradata or function call. This type of attack is based on the error handling when a contract sends money somewhere else using the `send()` function.

A smart contract should check for the failure or success of `send()`, but most of them do not. A send can fail when there is not enough gas to execute the default function at the contract that is receiving the money. Therefore, a `griefing` wallet could have a default function that takes an enormous amount of gas to run. This wallet could lock smart contracts by engaging with them so that when these contracts use `send()`, they fail and get disrupted.

There is a recommendation for the smart Ethereum Developer to `throw` if a call to `send()` fails, but this is still a problem because the `griefing` wallet could just lock the contract by making it throw all the time. Dimitris gave an example in lecture 4 where we have a contract that keeps an array of investors and adds a new investor if they invest more than the current smallest investment.

This griefing attack will not happen in the contract because I avoided using `send()` and `throw`. Nevertheless, it is worth mentioning that we are not using the "pull" design pattern because it is the smart contract the one which sends the money to the user, and does so with the "call" function (more on this in the *Reentrancy* section). It would be beneficial to create a separate `withdraw` function so that the users have to withdraw the money from the contract after they sell it.

2.3.2 Reentrancy

A `reentrancy` attack is a very powerful one that can be devastating for the contract. This attack can completely drain the smart contract of its ether, and it can also sneak its way into the code.

This can occur when one of the functions from the smart contract calls an external (untrusted) source before it resets/finishes/resolves the state of the contract. To give an example, a contract should update the balance of a user before sending a given amount to it; this way the contract prevents the `fallback` function of the receiving address making it give them more money. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

This is an attack that we have to be careful with because `customSend` uses the `call` function. It is recommended to stop using `.transfer()` and `.send()` in favour of `.call()`, but we have to take precautions because `.call()` does not have any check to mitigate reentrancy attacks. One of the recommendations in Solidity is to use the **checks-effects-interactions** pattern:

1. Perform checks e.g. who is msg.sender, did they send enough funds, are the arguments in range...

2. If all the checks passed, the function should now make effects to the state variables.

In my code I first check whether the smart contract can pay for the tokens and whether the user has enough funds before making any changes. It is worth mentioning again that it is often better to isolate each external call into its own transaction that can be initiated by the user. Better to let users to sell their funds using a function such as `sellToken` and make them retrieve their money by using a new `withdraw` function.

2.3.3 Front-running

Front-running basically consists on placing a transaction with some knowledge of a future one. The attacker can see a transaction and then place another one using this knowledge; but the attacker will do so with more gas so that the miner puts their transaction earlier than the first one.

For this smart contract it would be possible for a user to be a victim of a front-running attack when buying BdlToken tokens: When they are purchasing some tokens the owner of the contract might increase the price (if the smart contract has enough funds) with `changePrice` and increase the gas so that this transaction occurs before the purchase operation of the user. However, this will not be a huge problem that will make the user lost all their funds because if the user has sent the exact amount of money for buying the tokens (tokenPrice * numberOfTokensToBuy), the transaction will be reverted since the first check of the `buyToken` function (`require(msg.value >= amount * tokenPrice);`) will fail and the transaction will be reverted. This means there are two possible scenarios:

1. The transaction is reverted and the user has wasted the money which went to gas fees.

2. The user could have sent more money in the transaction so that he/she can buy the tokens even if the price is higher.

2.3.4 Other possible hazards

Some applications might required data to be private during some amount of time and they would have to be careful what source of randomness they use to prevent attackers reveal the data. For this contract we did not need to keep some information private so we need not use a *commitment scheme* as in coursework 2. Another possible problem could be unbounded operations, *underflows* and *overflows*; but after Solidity 0.8 the compiler automatically takes care of checking for overflows and underflows.

We might consider the scenario when the owner of the contract is malicious. In this case, the owner might want to steal users' money and he could do so by changing the tokenPrice to a very small value (or even 0) so that if the users try to sell their tokens they will not get any money and the funds will be kept in the contract. Nonetheless, the owner would need some way of accessing the smart contract funds, which he/she does not have. The owner could maybe add a function or link the contract to another smart contract which sends the contract's funds (`address(this).balance` to the owner's address.

2.4 Transaction history

I have already provided the transaction cost and hash of interacting with the smart contract in 2.2.1. However, I will provide now the deployment and transaction history of specifically doing what the coursework is asking for: buy, transfer, sell a token and double the token price after selling at least one token.

To begin with, I deploy the smart contract

- Transaction hash: 0x6ea36bd6595af5140158ba62d5f8ee8e9aba65501e55b927eb335ab2e4a2ac3d
- Transaction cost: 662,728 gas
- Address of smart contract: 0xAC5dF88146571fd4EA80BF09D84a8eda2C534c13

I buy one BdlToken at `tokenPrice = 1000 wei`.

- Transaction hash: 0xefe7530e459edaa755444538853a3d04c13d015c1bdac76d580e044cb05f9842
- Transaction cost: 70481 gas

Then I transfer that token in the contract. I decide to transfer it to address(0).

- Transaction hash: 0xffb0e4f1b69a4b22a1efa8474901b274958fb0493bad4f608a7c5803ac59c06a
- Transaction cost: 47,380 gas

Having transfered my BdlToken, now my balance of BdlToken tokens is 0 so I buy one more.

- Transaction hash: 0xd4075159ebfe970a9f16ef2911654551c6317642fbfe68b35fdbe3b64b968292
- Transaction cost: 53,381 gas

Then, I proceed to sell this token to the smart contract with the function `sellToken`. Note: I need to send at least 1 wei in the transaction due to the contract using the `customSend` function from `customLib` (which sends 1 wei to the owner of the library `customLib`) .

- Transaction hash: 0x1ce7f93bea702c1029337c01a0159fe4bb434b7277ed0fb89c03392d8d8a0a93
- Transaction cost: 52,895 gas

Last but not least, I need to double the price of the token. Remember that the smart contract checks whether it can buy all the tokens in circulating supply after having changed the price. So far there is only 1 token in supply (I just sold the second BdlToken I bought) which costs a total of 1000 wei. The contract's balance is 1000 wei because it bought my BdlToken, so I will not be able to double the price of the token.

In order to do this successfully, the contract needs to have 2000 wei (there is 1 token in circulating supply so if we double the price it will cost 2000 wei). Therefore, for the contract to have enough funds I will give it 1000 wei by calling the `buyToken` function with `amount = 0`, but with `1000 wei` sent with the transaction.

- Transaction hash: 0xa770a24082caee298f11028a47fe443c54d8cefed3adb00d0b1faf9a5e5803a4
- Transaction cost: 30,669 gas

Now I can indeed double the price of the BdlToken to `tokenPrice = 2000 wei`:

- Transaction hash: 0xd6e0674c27ff4ee4ddc62093988b6a5cd4bd3db7f7da687195cbfcbf4c5cc35b
- Transaction cost: 32,597 gas

This finishes the requested transaction history.

2.5 Code

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;
import './CustomLib.sol';

/** @title Custom Token contract. */
contract BdlToken {
    using CustomLib for uint;
    using CustomLib for address;

    /// Declared & Initialised here since we know the values beforehand
    /// and thus we reduce gas consumption
    address internal owner = msg.sender;
    uint public tokenPrice = 1000 wei;

    /// mapping containing addresses and the number of tokens they hodl
    mapping(address => uint) internal balances;
    uint internal totalTokens; /// variable to keep track of existing number of
         tokens

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

    /** @dev Updates balance of sender's tokens.
      * @param amount: uint of tokens to buy.
      * @return bool representing whether the transaction
      * was successful.
      */
    function buyToken(uint amount) payable public returns (bool) {
        uint totalPrice = amount * tokenPrice;
        require(msg.value >= totalPrice);
        /// update buyer's token balance
        balances[msg.sender] += amount;
        /// update total number of tokens
        totalTokens += amount;
        /// successful purchase hence emit event
        emit Purchase(msg.sender, amount);
        return true;
    }

    /** @dev Sends tokens from caller to a given recipient.
      * @param recipient: address to which the sender transfers the tokens.
      * @param amount: uint of tokens to transfer.
      * @return bool representing whether the transaction
      * was successful.
      */
    function transfer(address recipient, uint amount) public returns (bool) {
        require(balances[msg.sender] >= amount);
        balances[msg.sender] -= amount;
        balances[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    /** @dev Sells tokens to the smart contract.
      * @param amount: uint of tokens to sell.
      * @return bool representing whether the transaction
      * was successful.
      */
    function sellToken(uint amount) payable public returns (bool) {
      /// By using customSend we send 1 wei to the owner of customLib
      require(msg.value >= 1 wei);
      /// Without the 1 wei the user would receive tokenPrice wei for each
      /// sold token - 1 wei paid in customSend. Thus we add 1 wei extra.
      uint amountToPay = amount * tokenPrice + 1;
      /// contract needs to have enough balance to give the corresponding
      /// tokenPrice amount for each of the sold tokens
      require(balances[msg.sender] >= amount && getContractBalance() >=
          amountToPay);
      /// subtract tokens so they are "destroyed"
      totalTokens -= amount;
      /// subtract tokens from user's account
      balances[msg.sender] -= amount;
      /// use CustomLib for sending money to the seller
      bool success = amountToPay.customSend(msg.sender);
      emit Sell(msg.sender, amount);
      return success;
    }

    /** @dev Changes tokenPrice if called by the contract's owner and contract
      * has sufficient funds.
      * @param price: uint of the new price for the token.
      * @return bool representing whether the transaction
      * was successful.
      */
    function changePrice(uint price) public returns (bool) {
        /// Only the contract's owner can change tokenPrice
        require(msg.sender == owner);
        /// Add require statement so that "contract's fund suffice so that
        /// all tokens can be sold for the update price"
        require(address(this).balance >= price * totalTokens);
        tokenPrice = price;
        emit Price(price);
        return true;
    }

    /** @dev Returns token balance of sender.
      * @return uint representing the number of tokens.
      */
    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }

    /** @dev Returns balance of the smart contract.
      * @return uint representing the number of wei the smart contract has.
      */
    function getContractBalance() internal view returns (uint) {
        return address(this).balance;
    }

}
```

3 KYC Considerations and Token issuance

To implement the ”Know your Customer” process for verifying the identity of the customers we would require them to have a signature token which should have been issued by a trusted third party, by a certificate authority. The process of verifying the identity to give the signature token would be something similar to what centralized exchanges require their users to do before they are allowed to use all their functionality. As an example, Coinbase requires you to upload a picture of an identification document of yours and some more details before you can deposit and withdraw fiat.

Having said that, and with users having a way to obtain their signature token through a certificate authority, they would need to use such token every time they want to buy tokens from our smart contract. In our case, we want to associate an encrypted file with the `buyToken` operation. To do so, we would change the `buyToken` function to accept another argument `uint ciphertext` that we would check in the body of the function. In order to create the ciphertext that would help the smart contract verify the user's identity, the user would have to sign the identification document with the private key of their signature token. To improve the efficiency of the process, we would make the user sign the `hash` of the identification document with their private key.

`buyToken` will have to check whether the ciphertext provided is a valid one. We will not only need the hash of the identification document signed, but also the hash without signature. This is because the smart contract will use the public key of the user to verify the signed hash and it will have to check then whether it matches the hash of the document. These checks will be executed first in the function so that if they fail the transaction is reverted.

There are some problems which might arise here. Since the data stored in the blockchain is publicly available, we could have some user try to use another user's hashes to purchase tokens avoiding the KYC. To avoid this we could make the users use signing algorithms with timestamping; or add a nonce to the hash of the document before signing it.

The changes required to implement KYC cannot be completely made on-chain. We require the trust of a certificate authority outside of the blockchain (remember the blockchain is public and malicious users could try to use other users' documents) for verifying the identity of users. That is why Decentralised Exchanges are becoming more and more popular since governments are starting to include more and more regulation, which some users want to avoid by using DEXes such as Uniswap, Quickswap, Pancakeswap, Raydium, Orca...