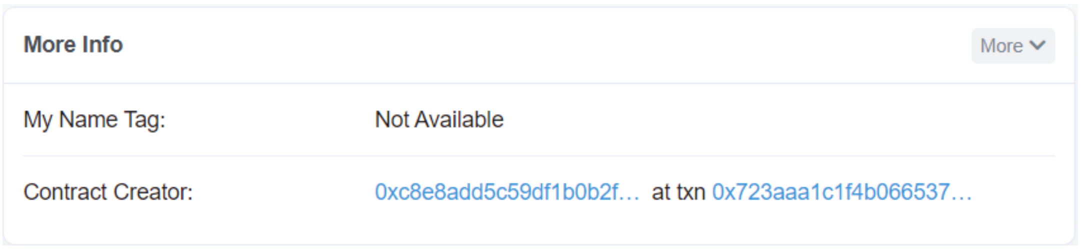
●Part 1 (10 points):

In this part, you will interact with a smart contract that has been deployed on Ethereum's testnet, Ropsten. You should call the "register" function at least once, with your student number as the second argument. Your report should detail how you managed to perform the transaction, detailing what key you used and how you found it.

Answer:
Key : actually...;)

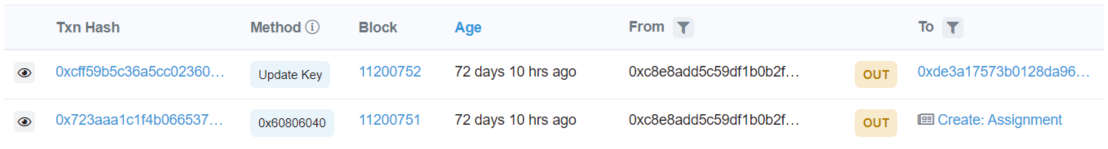
The way I found it:
I put the address of this contract to The Etherscan Website and selected to explore in Ropsten Testnet Network. Then I found the information of this contract as attached on this page:
https://ropsten.etherscan.io/address/0xde3a17573B0128da962698917B17079f2aAbebea



More Info

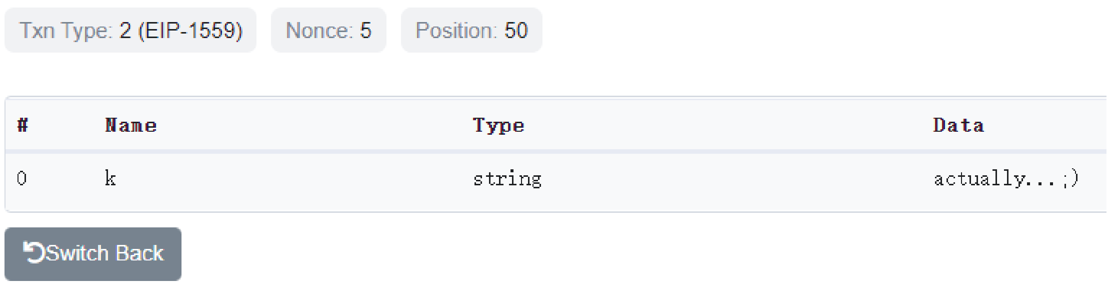
My Name Tag:	Not Available
Contract Creator:	0xc8e8add5c59df1b0b2f... at txn 0x723aaa1c1f4b066537...

Then in the information part we can find the address of the creator and the transaction of the creation. By reviewing the code of this contract, we can find that only the owner, i.e., the creator is allowed to set and update the key. Thus, by exploring the transactions history of the creator's address, we can find the last update that the creator made.



Txn Hash	Method	Block	Age	From		To
0xcff59b5c36a5cc02360...	Update Key	11200752	72 days 10 hrs ago	0xc8e8add5c59df1b0b2f...	OUT	0xde3a17573b0128da96...
0x723aaa1c1f4b066537...	0x60806040	11200751	72 days 10 hrs ago	0xc8e8add5c59df1b0b2f...	OUT	Create: Assignment

Then through the transaction details we can find the inputs arguments as below:



Txn Type: 2 (EIP-1559) Nonce: 5 Position: 50

#	Name	Type	Data
0	k	string	actually...;)

According to the source code, we can make sure that it's the latest key.

Performing the registration:
Firstly, I compile the code in remix IDE, then, switch the environment to Injected Web3, after that choose to interact with the contract at this address. Then, just call the register method, fix the key and UNN number, and click to call it.

●Part 2a: Smart Contract Programming Part II - Token (70 points):

● A detailed description of your high-level design decisions, including (but not limited to):
○ What internal variables did you use?
○ What is the process of buying/selling tokens and changing the price?
○ How can users access their token balance?
○ How did you link the library to your contract?

Answer:

1. Internal variables:

The internal variables I used are listed below: existAmount, balances, fundPool and locked.
The goal of including these variables is described in the comments.
Among them:
INITIAL_PRICE: used to record the initial price of this token.
existAmount: record the number of the existing tokens, increase when bought and decrease when sold.
balances: record the number of tokens in each address's account.
owner: record the address of the owner of this token contract, used to conduct access control.
FUNDPOOL: used to record the corresponding fundPool contract.
Locked: a bool variable used to resist re-entrancy attack.

```
//a immutable variable used to record the initial price
uint256 private immutable INITIAL_PRICE;
//a uint256 records the amount of the existed token
uint256 private existAmount;
//a mapping records the balances of users
mapping (address => uint256) balances;
//a address records the creator's address
address private owner;
//a address records the fundPool's address
FundPool private immutable FUNDPOOL;
//a bool variable to resist re-entrancy attack
bool private locked;
```

2. Process of buying/selling tokens and changing the price:

The process of buying and selling tokens is quite simple, user and customer can check the token price through the public variable **tokenPrice** whenever they wanna. If they choose to buy token, they can call the buyToken function, and fill in the number of token they want to buy. If they send sufficient ether, they get the token stored in their belances. It is almost the same procedure with selling token, they firstly decide how many tokens they want to sell, fill in it and call the function. If they did have enough tokens in their account, they would get ether from the fundPool contract.

When it comes to changing the price, it's a little bit different. First of all, let's make some assumptions. We assume that the goal of the creator is to raise funds, he wants to get as many ether as possible. Thus, he would not be motivated to just grab all the ether the first time someone buy them. Secondly, let's assume that our token is not just used as coin or note or any other stuffs like cash. We assume that our token could be regarded as one kind of proof of stake, you hold more, you have more power in our community. The goal of both the creator and the customer is not just to earn ether from the variation of token price. Of course, whenever you want to quit this game, you can sell all your tokens and just leave. However, if you choose to stay here and hold tokens, it means that you can receive other things such as the right to vote for decisions or strategies on how to use these ether. Thus, we assume that the selling of tokens must be after careful consideration. This makes sure that users would not sell their tokens instantly when they find the price increases compared with the price when they bought the tokens.

Based on these assumptions, I designed the change price function , which only allows the price of token to be set higher than its initial price. This is reasonable, with the collection of ether, each stake tends to be more and more expensive. At the meantime, when the price is changed, the owner must prepare enough money (not all) for the payment of part of the token to be sold.

3. Access token balance:

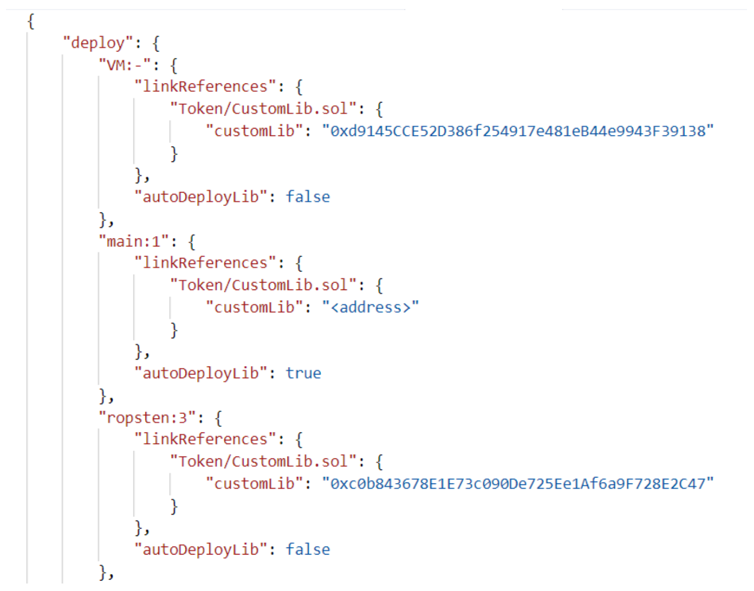
Users can access their balances whenever they call the getBalance method, it would return the balance based on their address.

4. Link the library to your contract:

There are several steps that I need to do in Remix to link the library to my contract:

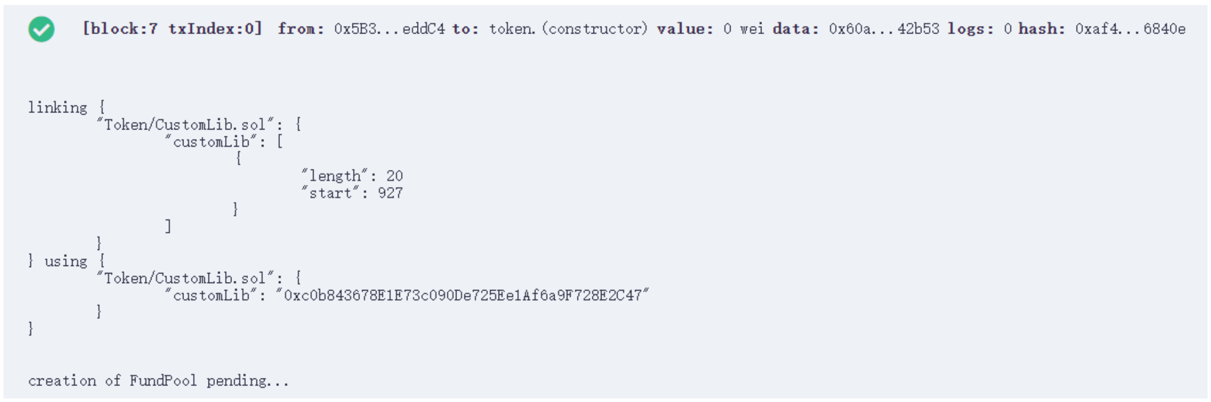
1. Use the import keyword to include the library into my contract
2. Allow the generation of json file in remix settings
3. Replace the "<address>" in VM and Ropsten with the lib's address in json file
4. Close the auto deploy library function
5. Deploy my contract on the Ropsten testnet.

The settings in json:



```
{
    "deploy": {
        "VM:-": {
            "linkReferences": {
                "Token/CustomLib.sol": {
                    "customLib": "0xd9145CCE52D386f254917e481eB44e9943F39138"
                }
            },
            "autoDeployLib": false
        },
        "main:1": {
            "linkReferences": {
                "Token/CustomLib.sol": {
                    "customLib": "<address>"
                }
            },
            "autoDeployLib": true
        },
        "ropsten:3": {
            "linkReferences": {
                "Token/CustomLib.sol": {
                    "customLib": "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
                }
            },
            "autoDeployLib": false
        },
```

The result is shown below:



```
[block:7 txIndex:0] from: 0x5B3...eddC4 to: token.(constructor) value: 0 wei data: 0x60a...42b53 logs: 0 hash: 0xaf4...6840e


linking {
        "Token/CustomLib.sol": {
                "customLib": [
                        {
                                "length": 20
                                "start": 927
                        }
                ]
        }
} using {
        "Token/CustomLib.sol": {
                "customLib": "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
        }
}

creation of FundPool pending...
```

Besides, another evidence that I truly linked to the contract is that I found that when we call the customSend function from the lib that deployed at 0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47, it transfers 1 wei to 0xC8e8aDd5C59Df1B0b2F2386A4c4119aA1021e2Ff instead of 0x8ec42d4D2CbAd10FfD90Ef8033AadFf3d25fbafB which is shown in the source code you provide.

5. Other high-level decisions:

Since the requirement that we are not allowed to add more public/external methods to this contract, we could not directly send ether to the balance of this contract (no fallback and receive functions as they are all external visibility), besides, for the same reason, we also could not use a deposit function to add more money for the token contract. The only way that we can take to send ether to this token contract is to make the constructor payable and send as more ether as possible at the creation stage. However, no matter how much ether we stored in the contract, if we keep increasing the price, there would be a time when we could not afford for the payment to users when they try to sell their tokens. The higher the price is set, the more ether we would lose.

Thus, based on these reasons, we could not rely on this token contract to save the ether. In this case, I decided to build a fundpool contract, and linked it to this token contract. In which case, the responsibility of the token contract would only be providing a layout for the users. It is the business logic layer. Our customers could interact with it through the interface, but it is not responsible for storing or transfering the ether.

The API of the fundPool contract is listed below:
Through the function modifiers realizing access control, certain function like register could only be called by the owner. The owner could use this function to register those addresses that he would like to link to this fundPool. Only those registed address could deposit ether in the fundPool and call it to transfer ether to other addresses.

```
address public owner
function register(address _address) public isOwner returns (bool isSuccessful)
function deposit() public payable registed returns (bool isSuccessful)
function transferMoneyTo(address recipient, uint256 amount) public registed returns
(bool isSuccessful)
function showBalance() public view returns(uint)
event Registration(address _address);
event Deposit(address sender, uint256 value);
event Transfer(address _to, uint256 amount);
```

● A detailed gas evaluation of your implementation, including:
○ The cost of deploying and interacting with your contract.
○ Techniques to make your contract more cost effective.
○ What was the gas impact of using the deployed library instance, compared to including its code in your contract?

Answer:

1. Gas evaluation:

The estimated gas for each operation for both the owner and user are listed below. The values in optimized row are got from the compiling optimization before deployment.

Table 1 Gas Costs of each operations

Operator	**Operations**	**Gas**	
		Unoptimized	**Optimized**
Owner	Deploying the fundPool contract	684730	475012
	Depositing ether into the fundPool	25282	24802
	Deploying the token contract	1249999	832706
	Register token contract's address in fundPool	47962	47383
	Change the token Price	36092	35100
Users	Buy tokens	84071	83010
	Sell tokens	68755	65899
	Transfer tokens	52420	51222

*Estimated in Ropsten environment

2. Cost effective:

The main techniques that used to save gas fee is enabling the compiling optimization during the compile stage. According to Table 1, it can be seen that the optimized bytecode would save us about 630,000 gas on deployment. Furthermore, for each operation that users would take, all the costs decrease by varying degrees.

It is worth to mentioned that Solidity compiler provides us totally different two way to optimize the contract, depending on the expected number of transactions that the contract would take. One way is to decrease the gas of the deployment, but the cost of calling function would increase, another way is that the deploying gas would increase but the subsequent calling of function would decrease its gas. Thus, it's obviously that since we are building a token contract and its corresponding fundPool, they are high likely to response to customers frequently, in this case the parameter " the expected number of runs" is supposed to be set as 1000 times during compiling.

Table 2 Gas costs in the comparison of different optimized way

Operator	**Operations**	**Gas**		
		2000 runs	**10 runs**	**difference**
Owner	Deploying the fundPool contract	492497	417441	75056
	Depositing ether into the fundPool	24802	24906	-104
	Deploying the token contract	898237	805171	93066
	Register token contract in fundPool	47383	47468	-85
	Change the token Price	35145	35193	-48
Users	Buy tokens	84080	84318	-238
	Sell tokens	70826	71014	-188
	Transfer tokens	51222	51302	-80

*Estimated in Ropsten environment

According to Table 2, we can find that by setting the runs to 2000 times, we need to pay about 160,000 more gas compared with the expected runs set as 10 times, but the buying, selling, and transferring function would cost about 500 gas lower. In this case, if our token contract has more than 320 transactions, we get started to save gas.

Besides the optimizations during compiling, there are also other techniques that used to make the contract more effective.

For instance, there must be an internal variable that used to records the balances of users, instead of using an array to conduct it, a mapping typed variable seems to be more effective, as we really do not care the order or index of these users. If we chose arrays, its read, rewrite, delete and insert operation would cost slightly higher than mapping, especially the insertion since arrays need to update the length all the time.

3. Gas impact of using the deployed library instance:

Compared with using the deployed library instance, if we choose to deploy a new one, it will take extra 236631 gas. Although it seems to be not very large, if the gas price is high, it may take quite a large amount of dollar/pound if we deploy it on the Ethereum main chain.

● A thorough listing of potential hazards and vulnerabilities that can occur in the smart contract and a detailed analysis of the security mechanisms that can mitigate these hazards.

1. Re-entrancy

Re-entrancy is quite a common hazard that occur in many smart contracts. In the token contract, the only place that would be attacked by re-entrancy would be the sellToken function. Two measures were taken to tackle with this kind of vulnerability. Firstly, follow the Check-Effects-Interactions Pattern to make sure that all the internal states/variables are changed first. Besides, a noReentrancy modifier is defined, which use a private bool variable *locked* to lock the state of the function when it is running.

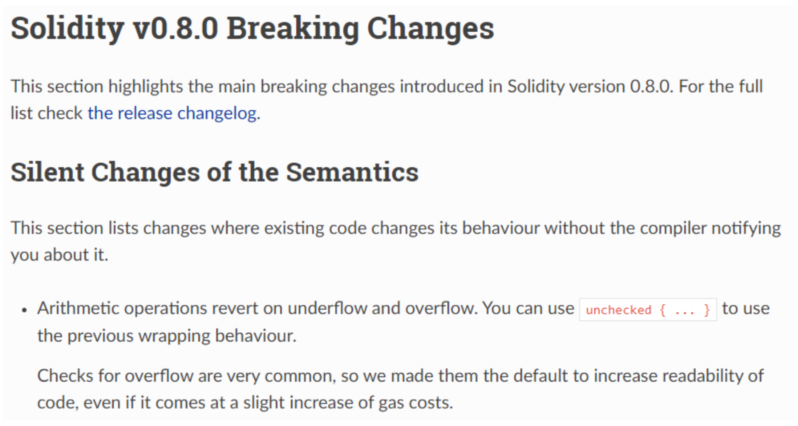
```
    modifier noReentrancy() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }
```

```
   function sellToken(uint256 amount) public noReentrancy returns (bool isSuccessful) {
           require(amount*tokenPrice > 1,
                   "Sorry,the transfer operation would take 1 wei as hand fee,"
                   " thus, you must make sure that the price of the sold token"
                   " is higher then 1 wei.");
           require(balances[msg.sender] >= amount,
                   "Sorry, you do not have enough token in your balance.");
           balances[msg.sender] -= amount;
           existAmount -= amount;
           bool success = FUNDPOOL.transferMoneyTo(msg.sender, amount*tokenPrice);
           require(success, "Transfer from fundPool failed.");
           emit Sell(msg.sender, amount);
           return true;
   }
```

2. Overflow and underflow

Overflow and underflow are also common hazards that may occur when the results of arithmetic operation reach the limitation of certain data type. The common solution is to use safaMath library or use the require statement to make sure that the calculation results are within the expectation. Such as require($a - b < a$), which are the same cases with other operations. However, solidity 0.80 has brought about some breaking changes, arithmetic operations revert on underflow and overflow now. This kind of changes aimed at increasing the readability of code, as the check for overflow and underflow were quite common, thus they were changed to default now.

On the other hand, if the contract is wrote based on solidity lower than 0.80, the use of SafeMath library or other measures are still necessary to prevent the overflow and underflow attack.



Solidity v0.8.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.8.0. For the full list check the release changelog.

Silent Changes of the Semantics

This section lists changes where existing code changes its behaviour without the compiler notifying you about it.

- Arithmetic operations revert on underflow and overflow. You can use `unchecked { ... }` to use the previous wrapping behaviour.

  Checks for overflow are very common, so we made them the default to increase readability of code, even if it comes at a slight increase of gas costs.

3. Malicious behaviour by creator

As I have mentioned in high-level decisions, in order to make sure that the token contract could always afford for all the token it sells, it's necessary to build a fundPool whose balance could be monitored by the owner. The owner could fill ether in the fundPool when necessary. In this kind of design, the token contract would hold a fundPool typed variable to receive the address of the deployed fundPool. However, this is the place where the creator could be malicious, if the address that linked to the so-called "fundPool" does not truly refer to a fundPool contract or firstly refers to a safe version and then switch to a malicious version which would directly transfer all the ether to the owner, it would result in an irreversible consequence. In this case, it's necessary for all the users to make sure that this address is linked to a safe version of fundPool before they decide to buy the token. Even though I have set this address variable to immutable, the customers still need to pay attention to this contract, and it would be better that the owner open its source code to all the users.

4. DoS

In current design, the token contract is not likely to be stuck, as it can deal with all the buying, selling, and transferring transactions at the same time. Each customer interacts with this contract independently, if someone's transaction fails, it does not influence other transactions. The only risky point is that, if the owner increases the price, but he does not have enough ether to afford the existed tokens, and if all the users try to sell their tokens, it will get stuck. But based on our assumption, since increasing price means lose ether, the owner is rational, he would not be motivated to do so to lose all the ether that he can manage to earn money in other places.

- **The transaction history of the deployment of and interaction with your contract.**

FundPool contract:

Addressed at:	0x0486e35e90E12224f68a81E66A167CE257762335
Created TX:	0xfa4dac8821192a4231a53e94955cce141734bbbf3b33ba83198bcb58ea21d676
Deposition TX:	0x54f2c19a1aae3016943aeb42d371d0835185795dde05885300e5c644d314fadf
Register TX:	0xb7863977a58b83baab5ff612db5334fde24a132d72ea25e8c69c69442ff67d65

Token contract:

Addressed at:	0xB5157c829b8743Fd7f3e28d77dEA7f75C9aEC706

Created TX: 0xe6d0c0ac7006171ea3568b9a37e3eeae3e45166b4ca9be2cfb8d5f5d1bff4cda
BuyToken TX: 0x7823a5f2d0f6cfbed937441b53ace988f18417f0debf7354e661ccebdfff8406
ChangePrice TX: 0x8ea13b9def1d4fcd1ca9f28250e41c3e298ee6e7de0065aae757e29d1c6974c2
SellToken TX: 0x2155eb5c42400261b191f6ec3d8681aa967068ee3ed088235b71c966b0ec6eaa
TransferToken TX: 0x4718ae73c2ea2806fa0d29ffd8cb2ae598deab07991c0c2242f2629f32e0c13f

● **The code of your contract.**

The code is attached below, you can also find it on my GitHub.

FundPool Contract:

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

pragma solidity >=0.8.0 <0.9.0;

import "./CustomLib.sol";
contract FundPool{

  address public owner;
  mapping (address => bool) registeredAddr;

  constructor (){
       owner = msg.sender; // assign the address of owner
       registeredAddr[msg.sender] = true;
       emit Registration(msg.sender);
   }

  modifier isOwner() {
       require(msg.sender == owner, "You are not authorized to operate this function.");
       _;
   }

   modifier registered() {
       require(registeredAddr[msg.sender],
       "You are not authorized to operate this function.");
       _;
   }

   event Registration(address _address);
   event Deposit(address sender, uint256 value);
   event Transfer(address _to, uint256 amount);

  function register(address _address) public isOwner returns (bool isSuccessful) {
```

```
            require(!registeredAddr[_address], "This address has been registered");
            registeredAddr[_address] = true;
            emit Registration(_address);
            return true;
  }

  function deposit() public payable registered returns (bool isSuccessful){
            emit Deposit(msg.sender,msg.value);
            return true;
  }

  function transferMoneyTo(address recipient, uint256 amount)
                                            public registered returns (bool isSuccessful) {
            require(address(this).balance >= amount,
                      "The fund pool does not have sufficient money now, please wait"
                      " for the owner to fill.");
            bool success = customLib.customSend(amount,recipient);
            require(success, "Transfer failed.");
            emit Transfer(recipient,amount);
            return true;
  }

  function showBalance() public view returns(uint){
            return address(this).balance;
  }

  function isRegistered(address _address) public view returns(bool){
            return registeredAddr[_address];
  }
}
```

Token Contract:

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

pragma solidity >=0.8.0 <0.9.0;

import "./FundPool.sol";
contract token{

  //a uint256 that defines the price of your token in wei;
  //each token can be purchased with tokenPrice wei
  uint256 public tokenPrice;
  //a immutable variable used to record the initial price
```

```
uint256 private immutable INITIAL_PRICE;
//a uint256 records the amount of the existed token
uint256 private existAmount;
//a mapping records the balances of users
mapping (address => uint256) balances;
//a address records the creator's address
address private owner;
//a address records the fundPool's address
FundPool private immutable FUNDPOOL;
//a bool variable to resist re-entrancy attack
bool private locked;

constructor (uint256 initialPrice, address _fundPoolAddr) payable{
     owner = msg.sender; // assign the address of owner
     INITIAL_PRICE = initialPrice;
     tokenPrice = initialPrice;
     FUNDPOOL = FundPool(_fundPoolAddr);
  }

  // Modifier to check that the caller is the owner of the contract.
  modifier onlyOwner() {
     require(msg.sender == owner, "You are not authorized to operate this function.");
     _;
  }

  modifier noReentrancy() {
     require(!locked, "No reentrancy");
     locked = true;
     _;
     locked = false;
  }

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

/* a function via which a user purchases amount number of tokens
   by paying the equivalent price in wei;
   if the purchase is successful, the function returns a boolean value (true)
   and emits an event Purchase with the buyer’s address and the purchased amount
*/
function buyToken(uint256 amount) public payable returns (bool isSuccessful) {
          require(FUNDPOOL.isRegistered(address(this)),
          "Sorry, the token contract has not been registered with FundPool.");
          require(msg.value >= tokenPrice*amount,
          "Insufficient value, please check current price then send enough value.");
          balances[msg.sender] += amount;
          existAmount += amount;
          bool success = FUNDPOOL.deposit{value:msg.value}();
          require(success,
          "There is something wrong with the link between token contract and fundPool"
          "please wait for the owner to handle it.");
          emit Purchase(msg.sender,amount);
          return true;
}

/* a function that transfers amount number of
  tokens from the account of the transaction’s sender to the recipient;
   if the transfer is successful, the function returns a boolean value (true)
   and emits an event Transfer, with the
   sender’s and receiver’s addresses and the transferred amount
*/
function transfer(address recipient, uint256 amount)
                                                public returns (bool isSuccessful) {
          require(balances[msg.sender] >= amount, "You do not have enough balance.");
          balances[msg.sender] -= amount;
          balances[recipient] += amount;
          emit Transfer(msg.sender,recipient,amount);
          return true;
}

/*   a function via which a user sells amount number of tokens
     and receives from the contract tokenPrice wei for each sold token;
     if the sell is successful, the sold tokens are destroyed,
     the function returns a boolean value (true) and emits an
     event Sell with the seller’s address and the sold amount of tokens
*/
function sellToken(uint256 amount) public noReentrancy returns (bool isSuccessful) {
          require(amount*tokenPrice > 1,
          "Sorry,the transfer operation would take 1 wei as hand fee,"
          " thus, you must make sure that the price of the sold token"
          " is higher then 1 wei.");
          require(balances[msg.sender] >= amount,
          "Sorry, you do not have enough token in your balance.");
          balances[msg.sender] -= amount;
          existAmount -= amount;
          bool success = FUNDPOOL.transferMoneyTo(msg.sender, amount*tokenPrice);
```

```
            require(success, "Transfer from fundPool failed.");
            emit Sell(msg.sender, amount);
            return true;
            }

     /* a function via which the contract’s creator can change the tokenPrice;
      if the action is successful, the function returns a boolean value (true) and emits
      an event Price with the new price (Note: make sure that, whenever the price changes,
      the contract’s funds suffice so that all tokens can be sold for the updated price).
     */
      function changePrice(uint256 price) public onlyOwner returns (bool isSuccessful) {
            require(price >= INITIAL_PRICE, "You could not change the price "
                                            "to be lower than its initial price.");
            require(FUNDPOOL.showBalance() >= existAmount * price,
            "The fundPool do not have sufficient banlance "
            "to pay for all the existed token according the price to be changed.");
            tokenPrice = price;
            emit Price(price);
            return true;
      }
   /*a view that returns the amount of tokens that the user owns*/
      function getBalance() public view returns (uint256){
          return balances[msg.sender];
      }
   }

```

Part 2b: KYC Considerations and Token issuance (20 points)

Suppose that the smart contract issuer has to comply with regulation related to KYC (“know your customer”), where token issuance can happen only in case some identification document has been provided. Therefore, you want to associate with each buyToken operation an encrypted file that corresponds to an identification document of the token recipient. Describe in your report a way you can use a public-key encryption scheme to incorporate such a ciphertext in the buyToken operation, as well as any other changes needed in the above API. Is it possible to implement this process completely on-chain? Describe in detail the steps and tools (using any relevant material from the lectures) needed to implement this.

Answer:

Generally, I do not think that all these operation could be implemented completely on chain. Firstly, it’s not a good idea to save the identification files which may be an image or a documentation on chain, as the storage cost would be quite high. Secondly, since there do not exist truly “private” data on chain, if we request users to directly send or upload their unencrypted identification file, it may result in severe privacy disclosure. Thus, we need to take some measures to make sure that only we, the creator of this token contract has the access to these identification file, and at the mean time we can make sure that the information is sent by certain user, i.e., the unforgeability is required.

To be specific, since we need to use a public-key encryption scheme, which is also known as asymmetric cryptography. We need to reply on some tools to generate key pairs, where the public key is open to anyone, and the private key is used to decrypt it. Currently the most used public key algorithm should be RSA, however, RSA is a resources expensive algorithm, as a rule of thumb, you can only encrypt data as large as the RSA key length. Thus, it's often not possible to encrypt large files with RSA directly. Instead, it's quite common to use RSA to encrypt a password and encrypt the file with the derived password and then send both the file and the encrypted password to receiver.

On the other hand, though the RSA algorithm and the public key encryption solves the problem of transmitting the information safely, it could not be regarded as a proof to the source of the sender. It means that if someone claimed that he/she to be the sender, we could not verify it. In this case, we need to employ another tools, the digital signature to guarantee the unforgeability.

Before taking steps, we need to make some clarification and assumptions. Firstly, we need to assume that the KYC organization have the capability to verify whether the identification files are valid. In other word, they must be able to locate the identity of any person based on the combination of some types of identification files, for instance, the photograph, the biometric information and the home address. This means that actually, the KYC organization must have held all these relevant identification information of residents in their database before they ask us to follow these regulations. We, the contract provider should not take the responsibility of logging new type of identifications. We only receive those valid type of identifications files such as BRD card, passport or any documentation else.

Supposing that the creator of contract provides a public key Pk, for users to encrypt their information, and keep a private key to decrypt the files they received. Besides, each user also need to generate a pair of keys, the sign key, sk and the verification key, vk. They should keep the sk privately, but the verification key could be opened to everyone.

Based on above analysis and assumptions, certain steps could be taken to realize the whole procedure:

1. The contract creator provides its public key, pbk, for uers to conduct encryption.
2. Users generate their *sk* and *vk*.
3. Users use algorithm to Generate a 64-byte random password (or any other length they want to)
4. Users use the password encrypt his/her identification files
5. Users upload their encrypted files on a third-party cloud storage service provider (such as google drive) and generate a link.
6. Users generate a message, *m* based on this password, the cloud drive link, and their address. (This is important to resist the front running attack).
7. Users use contract's public key, pbk to encrypt this message and generate the encrypted message m'.
8. Users use their sk to sign on the message, and generate a signature σ, which is equals to Sign(m,sk).
9. (On chain) Users call the buyToken function which receives new parameters including m', σ, and vk to contract. This would emit an event and logging these data.
10. Creators download the log and use their private key to decrypt m' and get m

11. Creator use users' vk to decrypt σ and comparing the results with hashed m' to make sure the identity cannot be forged
12. Creators need to check that whether the msg.sender's address is equal to the address encrypted in the massage to avoid front running attack.
13. After that, creators can use the link and password in massage to get and decrypt the identification files
14. Then creators can share these files with the KYC authority and request verification.
15. If these files get verified, creators now need to assign the tokens to users' balances
16. If not, the ether that users sent would be deposit into their balance, they can call the withdraw function to get back their ether.

The changes needed include, firstly, we need to change the pattern of buying tokens, users need to send the extra encrypted identification data now. Besides, they could not get their tokens immediately, they need to wait for manually assignment after their identity get verified.

Thus, we may need to add two more function to the API, one is the assignToken, which could only be called by the administrator or owner. The second one is withdraw, for uses to get back their ether which they sent by failed to buy token.

Furthermore, if we want to realize a more comprehensive access control, we'd better add a new mapping(address => bool) variable named "verified" to record whether certain address has been identified. Besides buying behaviour, transferring may also need to be restricted. We can add a new statement in transfer function and requires that the recipient's address is also identified.

Actually, from my point of view, comparing with verifying the identity when users are buying or transferring token, it would be better that we provide a register or verify function and ask all the users to call them first if they want to make any transactions with this token contract. By doing so, we can add a new modifier named "identified" thus we can control the access easier.