School of Informatics
Blockchains and Distributed Ledgers

Assignment 3

Erodotos Demetriou (s2187344)

Part 1

For Part A of this assignment, we are required to interact with a smart contract deployed on the Ethereum Ropsten network. In more detail, we should call the register method, giving input arguments a secret key and our student number. Since Ethereum is a public blockchain, everything can be accessed on-chain. Hence, the following script was used to retrieve the secret key from the smart contract's storage.

```
1 const Web3 = require('web3');

3 var web3 = new Web3(new Web3.providers.HttpProvider('https://ropsten.infura.io/v3/0
      ab1814012ad4231965d67bf98a40b1a'));

5 var contract_address = '0xde3a17573B0128da962698917B17079f2aAbebea';

7 web3.eth.getStorageAt(contract_address, 1).then(result => {
8     console.log(web3.utils.hexToString(result));
9 });
```

The found key is: "actually...;)". We imported the smart contract source code and address into *Remix* and finally called the register function with "actually...;)" and "s2187344" as inputs.

Transaction Hash : 0xc6ebbfea38258b115d07d5826667dfdd2445a5cbe277f0c3e18fa5a294ef3d32

Part 2 A

Smart Contract High-Level Description

Part 2 A of this assignment regards implementing and deploying a custom token. Its requirements include allowing users to exchange ETH for tokens, sell their tokens back for ETH, and transfer tokens between accounts. Also, the users can view their balance by calling the smart contract. Besides that, the token owner can change the token's price if there is enough balance in the smart contract.

To achieve that, we introduced the concept of liquidity providers. Liquidity providers are individuals willing to deposit their ETH to the smart contract and receive tokens as rewards whenever a new buyer purchases the token in the future. Next, we provide a real-world example of using this mechanism to understand its functionality better.

Figure 1 illustrates the smart contract's life cycle. In step A, the owner (address A) deploys the smart contract providing 100 Wei. This automatically registers him as the first liquidity provider. Next, in step B, we can observe a buyer (address B) conducting a buy request to the smart contract. For instance,

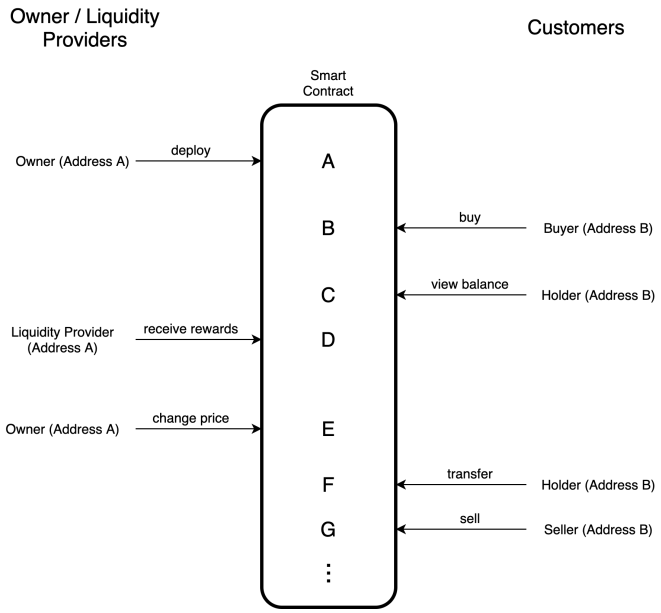

Figure 1: Smart Contract Execution Workflow Example

he wants to buy 30 tokens priced at 10 Wei each. Hence, he has to send 300 Wei to the smart contract and provide the number of tokens as an input parameter. We check for the validity of the input data, and if everything is accurate, we proceed. 10% of the purchased tokens is contributed into a rewards pool while the customer receives 90%. We have to mention that for simplicity objectives in calculating the percentage, we allow purchases of multiples of 10. Later, in step C the buyer can view its balance by calling the getBalance() method of the smart contract. At this point (D), a liquidity provider can receive rewards from the rewards pool. He will get a portion of tokens according to the fraction of liquidity he provided to the total liquidity. Each provider can increase the rewards he receives by offering more liquidity.

Nonetheless, we have to note that because solidity does not support floats, we restrict the amount a liquidity provider can offer to multiples of 100 Wei. Further, if there is only one token in the rewards pool and two liquidity providers own 50% of it each, the smart contract will not allow them to receive rewards because of an impossible division of $1/2$. They can receive their rewards later when the pool balance allows a fair distribution.

Now, suppose we want to change the price of our token. The owner can allow this action in step E. The only constrain of this transaction is that the contract's balance should be greater or equal to the product of circulating tokens and the new price.

Moreover, a token holder can transfer its tokens to another address. This is depicted in step F, where the user calls the smart contract providing the receiver's address and the amount of tokens.

Finally, any token holder may sell his tokens to the smart contract for ETH considering the current token price in Wei. The sellToken function does not directly transfer ETH to the user. Instead, a custom library deployed on the Ropsten network is used. One of the main challenges of this assignment was to link an already deployed library into our smart contract. This was impossible to achieve through *Remix* online editor. Thus we used *Truffle* smart contract development framework to achieve the desired outcome. Practically, we compiled the customLib.sol locally to generate a reference ABI and afterward import the library into our Token.sol contract. Since we needed to use an already deployed instance of the library, we specified its address in the migrations javascript file and used the link method to reference it. There follows the deployment and link code for the operation as mentioned earlier and a detailed overview of

smart contract's variables, functions and events.

```
const Token = artifacts.require("Token");
const CustomLibrary = artifacts.require("customLib");

module.exports = function(deployer, accounts) {
    CustomLibrary.address = "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
    deployer.link(CustomLibrary, Token);
    deployer.deploy(Token, { value: "100" })
};
```

Variables

address owner: This variable is set on deployment, and stores the smart contract's owner address.
uint256 contractBalance: This variable stores the number of Wei deposited into the smart contract.
uint256 circulatingTokens: This variable stores the number of circulating tokens.
mapping balances: This is a mapping variable maintaining a one to one correspondance between a user address and the number of tokens it holds.
mapping liquidityProviders: This is a mapping variable maintaining a one to one correspondance between a liquidity provider user address and the amount of liquidity it provided in *wei*.
uint256 numberOfProviders: This variable holds the number of total liquidity providers.
uint256 totalLiquidity: This variable holds the amount of total liquidity provided during the complete life cycle of the smart contract.
uint256 rewardsPool: This variable stores the amount of tokens that will be available to liquidity providers as rewards for their services.
uint256 tokenPrice: This variable stores the current token price in Wei.

Events

event Purchase: Whenever a token purchase is fulfilled, an event stating the buyer's address and the purchased amount, is emitted.
event Sell: Whenever a token holder sells its tokens back to the smart contract, a sell event is emitted. This event states the seller address and the amount of tokens sold.
event Transfer: Whenever a token holder transfers its tokens to an other address a transfer event is emitted. This event states the sender and receiver adddresses, as well as the transfered amount of tokens.
event Price: This event is emitted whenever the smart contract owner changes the token price.

Functions

constructor(): This is the smart contract constractor. It is called whenever the code is deployed for the first time, and sets the contract owner, as the initiator of the transaction. The smart contract constructor requirement is that the owner provides 100 Wei as initial liquidity. In other words, he becomes the first liquidity provider.
buyToken(): Whenever a buyer wants to exchange his ETH for our token has to call this function. In more detail, he needs to provide as input the number of tokens that he desires to purchase and send the proportional amount of ETH, considering the token price which is in Wei.
transfer(): This function can be called by a token holder that desires to transfer a specific amount of tokens to an other address.
sellToken(): This function enables a token holder to exchange his tokens with ETH. He can provide the number of tokens he wants to exchange and subsequentyl he will receive the according value of ETH considering the current token price.
changePrice(): This function can be called only by the smart contract owner, to change the token price. Its execution will only succed if there is enough liquidity to pay token holders if they decide to sell their token simultaneously.
getBalance(): This function returns the balance of tokens that are in the posetion of the fucntion's caller.

provideLiquidity(): This is an internal function that can be triggered through the fallback function. It receives an address and the amount of provided liquidity. Subsequently, it registers this user as a liquidity provider.
getReward(): This is an internal function that can be trigered through the fallback function and only if the caller is a registered liquidity provider. Its execution results into receiving a proportion of the reward tokens from the reward pool.
fallback(): This function can be used with different arguments and can serve different purposes. If it is called providing some ETH, registers the user as a liquidity provider; else, it can be used by liquidity provider to receive its rewards.

Gas Costs Evaluation

This section measures and evaluates the gas cost we expect to have when deploying and interacting with the smart contract.

Gas costs appear in the following table. The smart contract owner has to pay 1,389,915 gas units to deploy the code. This is a high amount of gas, making the smart contract expensive to deploy initially.

Gas Costs

Contract Owner Fees	
Contract Deployment	1,389,915

Customer Fees	
buyToken()	65,102
sellToken()	62,875
transfer()	52,328
changePrice()	34,639
getBalance()	None

Liquidity Provider Fees	
provideLiquidity()	42,083
getReward()	35,110

Regarding the smart contract customers interactions, the fees are much lower. Namely, when buying some tokens, the gas required is 65,102, while the invocation of sellToken function costs 62,875. We can observe that buying and selling tokens from and to the contract yields approximately the same overhead. Next, if a token holder transfers its tokens to another address, he can use only 52,328 gas units, which is relatively cheap. Retrieving the available balance of tokens for a specific address occurs at no expense.

Besides customer invocations, we have liquidity providers interacting with the smart contract. Providing liquidity costs 42,083 gas units. When designing the provide liquidity functionality, our goal was to keep it as cheap as possible to motive liquidity providers to back the token sale. We can argue that this has partially been achieved since depositing funds is cheaper than buying or selling tokens. Furthermore, when a liquidity provider requests its rewards, the costs are even cheaper at 35,110 gas units. Nonetheless, a liquidity provider will amortize its investment with more profits since 10% of each token purchase goes to him.

Potential Hazards and Vulnerablities

Developing a Smart Contract can always be challenging. This is because, on a public-permissionless blockchains, everything is observable by everyone. This fact makes it difficult when it comes to securing users' data. Besides that, a developer has to be alert to write code that is attack-resistant. Smart contracts expose a broad spectrum of vulnerabilities, enabling an adversarial entity to exploit them for his interest.

When developing our custom token, we took into consideration possible attacks. The following list presents vulnerabilities and mechanisms to moderate them.

DoS(Denial of Service) - Griefing: An attacker attempts to make a Smart Contract get stuck when executed. In the case of our implementation, we don't face such problems. The nature of the code does not allow the contract to get stuck. All functions are atomic, which means that they will complete or fail. The only scenario in which we might face grief in our smart contract is when there is insufficient liquidity to change that token price. In other words, if there are not enough liquidity providers, the contract owner won't be able to increase the token price frequently. Similarly, if don't have buyers, liquidity providers won't receive rewards and thus lose their initial investment.

Mitigation: To mitigate this issue, we require that the contract owner provide initial liquidity to the smart contract by becoming the first liquidity provider.

Re-Entrancy: An attacker might try to take advantage of the smart contract sellToken function by executing a re-entrancy attack. In more detail, when he invokes a sellToken transaction, he can drive his transaction to a malicious fallback function on another Smart Contract that can recursively call again and again the sellToken function, trying to get more ETH.

Mitigation: In order to avoid such unpleasant attacks, we execute the code of the smart contract in a particular way. When there is a withdrawal invocation, we check some constraints to ensure that the transaction sender is allowed to withdraw ETH. After that, we will pass any updates to the state of the smart contract and eventually make the call that sends the requested ETH to the recipient. It is essential to make the ETH transfer after changing the smart contract state. If a reentrancy attack occurs on the next transaction, it will be stopped because function constraints will evaluate the transaction according to the updated state.

Front-Running: This attack happens on the Miner level. An attacker might clone your transaction and put a much higher gas limit on it. This results in including his transaction in the next block instead of yours. This might be inconvenient when a buyer tries to buy some tokens priced at 10 Wei, and the smart contract owner front-runs him to increase the price. Further, we have to note that our token has no limited supply, and thus, there is no meaning in front-running its purchase. Everyone can buy any amount of tokens.

Mitigation: There is no straightforward solution to front-running since the problem lies at the transaction mining level. For our custom token, front-running will not have a significant effect.

Overflow/Underflow: Our custom token does not face this problem since it is coded using version 0.8.0 of Solidity, which automatically resolves any issues of over/under flow and aborts the execution of the smart contract.

Randomness Source Exposure: Our custom token does not involve using any random value.

Delegation: In our smart contract, we used a custom library to send funds to an address when a sale occurs. We assume that this library is safe as the assignment authors have provided it.

Other good practices:

- Use *call()* instead of *transfer()* or *send()*. Using *call()* might be insecure, but *transfer()* and *send()* can forward only 2300 gas. In the future gas costs might change, and 2300 gas might not be enough. As a result, using *call()* properly is the right approach. We use a custom library that uses call() to send ETH; thus, we check its output correctness, as shown in the example below.

```
1          (bool success = customLib.customSend(amount * tokenPrice, msg.sender);
2          require(success, "Transfer failed!");
```

- A liquidity provider might want to receive its rewards. For this part of the implementation, we decided it is safer for the smart contract to have an external user initiate the reward distribution, instead of pushing the tokens into his address. In other words we have them pulling their rewards.
- As for our fallback function, we keep it simple, allowing only two operations. It can only be used to provide liquidity or receive rewards. Any other operation on the smart contract's fallback functions will result in a deadlock.
- Moreover, it is an excellent practice to emit events when a critical transaction is complete. Hence, we emit events whenever a token sale, purchase, or transfer occurs.

Smart Contract Deployment and Execution History

Owner Address: 0x1959f7433D617977e144cC3Bb722D794384D562B

- **Smart Contract deployment**
  TX Hash: 0x5ba0d9a48485c1400f72bf2d0482b869f44935ceb38cbb8a720a5d66c391ece0
  Contract Address: 0x8252e289f6ef096CCb0647322102ac5459D1Df49
- **Buy Tokens**
  TX Hash: 0xeaf2f20e09dc42dbf743d6edd7618e831f3ce195967ccb4624b4b6dfa395ab88
- **Get Rewards**
  TX Hash: 0x7aef6df0c8b4a3d8f7685c3e7cc2f33117addeea284d0fd940abbab9e17c60b7
- **Change Price**
  TX Hash: 0xae26b213a00cd543b3a57e97d9f0626b18812df69fb10cc843432302442f122a
- **Transfer Tokens**
  TX Hash: 0xfa2a9b619d58b934957503f8afdd08ceafcd71e228d7d27fc357c8ad4b3c1976
- **Sell Token**
  TX Hash: 0x53be3484f8fe51bd58bc5f4aec9f0ae45711b7b8924c23f70f512ba21f0c5aa1
- **Provide Liquidity**
  TX Hash: 0x43f228412eb74dc55e40c95f5f020e4efe3bdc4b6eef91dda21a2041f08dd002

Implementation Code

```
pragma solidity >=0.8.00 <0.9.0;

import "./customLib.sol";

contract Token {
    address private owner;
    uint256 private contractBalance = 0;
    uint256 private circulatingTokens = 0;
    mapping(address => uint256) private balances;
    mapping(address => uint256) private liquidityProviders;
    uint256 private numberOfProviders = 0;
    uint256 private totalLiquidity = 0;
    uint256 private rewardsPool = 0;
    uint256 public tokenPrice = 10;
    event Purchase(address indexed buyer, uint256 amount);
    event Sell(address indexed seller, uint256 amount);
    event Price(uint256 price);
    event Transfer(
        address indexed sender,
        address indexed receiver,
        uint256 amount
    );
    constructor() payable {
        require(
            msg.value == 100,
            "As the contract owner you are required to provide 100 wei liquidity"
        );
        owner = msg.sender;
        provideLiquidity(owner, msg.value);
    }

    function buyToken(uint256 amount) public payable returns (bool) {
        require(
            msg.value == amount * tokenPrice,
            "Your funds are not sufficient"
        );
        require(
            amount % 10 == 0,
            "The purchased quantity should be multiple of 10!"
        );
        circulatingTokens += amount;
        contractBalance += amount * tokenPrice;
        balances[msg.sender] += (amount * 90) / 100;
        rewardsPool += (amount * 10) / 100;
        emit Purchase(msg.sender, amount);
        return true;
    }

    function transfer(address recipient, uint256 amount) public returns (bool) {
        require(balances[msg.sender] >= amount, "Not enough balance");
        balances[msg.sender] -= amount;
        balances[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    function sellToken(uint256 amount) public returns (bool) {
        require(amount >= 1, "Provide positive amount of tokens!");
        require(amount <= balances[msg.sender], "Not enough balance!s");
```

```
60          require(amount > 0, "Not enough balance!");
61          circulatingTokens -= amount;
62          balances[msg.sender] -= amount;
63          contractBalance -= amount * tokenPrice;
64          bool success = customLib.customSend(amount * tokenPrice, msg.sender);
65          require(success, "Transfer failed!");
66          emit Sell(msg.sender, amount);
67          return true;
68      }

70      function changePrice(uint256 price) public returns (bool) {
71          require(
72              msg.sender == owner,
73              "This function is restricted to the contract's owner"
74          );
75          require(
76              contractBalance >= circulatingTokens * price,
77              "Not enough liquidity"
78          );
79          tokenPrice = price;
80          emit Price(price);
81          return true;
82      }

84      function getBalance() public view returns (uint256) {
85          return balances[msg.sender];
86      }

88      function provideLiquidity(address provider, uint256 amount) internal {
89          liquidityProviders[provider] += amount;
90          totalLiquidity += amount;
91          contractBalance += amount;
92          numberOfProviders += 1;
93      }

95      function getReward() internal {
96          require(
97              liquidityProviders[msg.sender] >= 100,
98              "You are not a liquidity provider"
99          );
100         require(
101             rewardsPool % numberOfProviders == 0,
102             "You can not receive rewards now (unfair division)"
103         );
104         uint256 allowance = (liquidityProviders[msg.sender] / totalLiquidity) *
105             100;
106         balances[msg.sender] += (rewardsPool * allowance) / 100;
107         rewardsPool -= (rewardsPool * allowance) / 100;
108     }

110     fallback() external payable {
111         if (msg.value > 0) {
112             provideLiquidity(msg.sender, msg.value);
113         } else {
114             getReward();
115         }
116     }
117 }
```

Part 2 B

KYC, also known as "know your customer", is common practice among financial institutions when a new client enters their business. They use KYC to verify the client's identity and evaluate the risks of maintaining a business relationship. A straightforward KYC methodology might include submitting an identification document such as a birth certificate, proof of address of a legitimate utility bill, and a recent face photograph of the client. Within the context of Part 2 B of this assignment, we need to describe such a mechanism for our custom token sale. Further, we will argue if this is feasible to implement on the blockchain.

It is more than evident that having many customers yields a requirement for storing thousands of documents. The Ethereum blockchain is not the best option for storing documents. Foremost, we can not directly store multimedia such as pdf files and images on-chain. We should convert them into an intermediate text form such as base64 encoding. This is going to add computational overhead. Moreover, storing 1KB of data requires approximately 640K gas which is not cheap. Besides that, Ethereum is a public permissionless blockchain, thus storing KYC documents on-chain can violate the customers' privacy. Hence in our KYC proposal, we attempt to instrument a hybrid solution using both blockchain and a centralized server to process new customer requests. The main idea of our implementation is that a new customer submits its documents on the server for approval while publishing its KYC identification fingerprint on-chain. Afterward, the smart contract owner reviews the submitted documents off-chain, and if they are valid, he can approve them and enable the client to purchase tokens on-chain. There follows a more thorough explanation.

When designing the KYC mechanism, the foremost priority was user data privacy, and gas efficiency and fairness. For this reason, we adopt a public-key cryptography scheme to encrypt identification documents and sign them. To achieve that, we introduce a central trusted authority that keeps the public keys of both contract owner and client. When a new customer wants to initiate a KYC procedure, follows the steps below, as depicted in Figure 2

1. Clients and the contract owner publish their public keys to the trusted authority.
2. Clients and the contract owner retrieve public keys of the entity they desire from the trusted authority.
3. The server publishes a Nonce that encrypts using the client's public key and sends it to the client, as a challenge, to include in its subsequent KYC request. The client decrypts the Nonce using its private key.
4. The user prepares a PDF file, including its identification documents. We call this, Data. Next, the user concatenates Data and the received Nonce and encrypts them using the server's public key, producing a Ciphertext. Then he signs the Ciphertext by hashing it and encrypting it using his private key.
5. In step 5, the user forwards the signed Ciphered text to the server while publishing its signature/fingerprint on-chain.
6. The server receives the signed Ciphertext and proceeds to validate it. Mainly, we check for Nonce validity and Data (documents) validity and signature.
7. We retrieve the on-chain recorded signature and perform a peculiar 3-way check, comparing the submitted to server signature with the recalculated Ciphertext Hash and the signature submitted on-chain.
8. If everything is valid, the contract owner can proceed and approve the client's requests to purchase tokens from the smart contract.

We decided to introduce the concept of a nonce to avoid replay attacks when submitting the identification documents to the server. Further, we have both client and owner interacting with the smart contract, splitting the gas costs of completing the KYC process. This approach provides gas fairness and transparency to the client, ensuring that its digital fingerprint/signature is correct. This is important since that digital fingerprint associates the client with an encrypted form of its KYC documents.

Indubitably, we have to extend our smart contract's public API to support the mechanism above. This hybrid KYC tool requires very few changes to the smart contract, narrowing the gas costs required to implement it. In more detail, we need the following function and variables.

```
1       mapping(address => KYC) public kyc;
2       struct KYC {
3           bytes64 finger_print ;
4           bool approved;
5       }
6       function setFingerPrint(bytes64 finger_print) public returns (bool);
7       function getFingerPrint(address client_address) public returns (bytes64);
8       function approveClient(address client_address) public returns (bool);
```

Finally, we need to add a requirement statement into the buyToken function to check if the client's address is approved.

```
1       require(kyc[msg.sender].approved == True, "Your are not allowed to buy tokens")
            ;
```

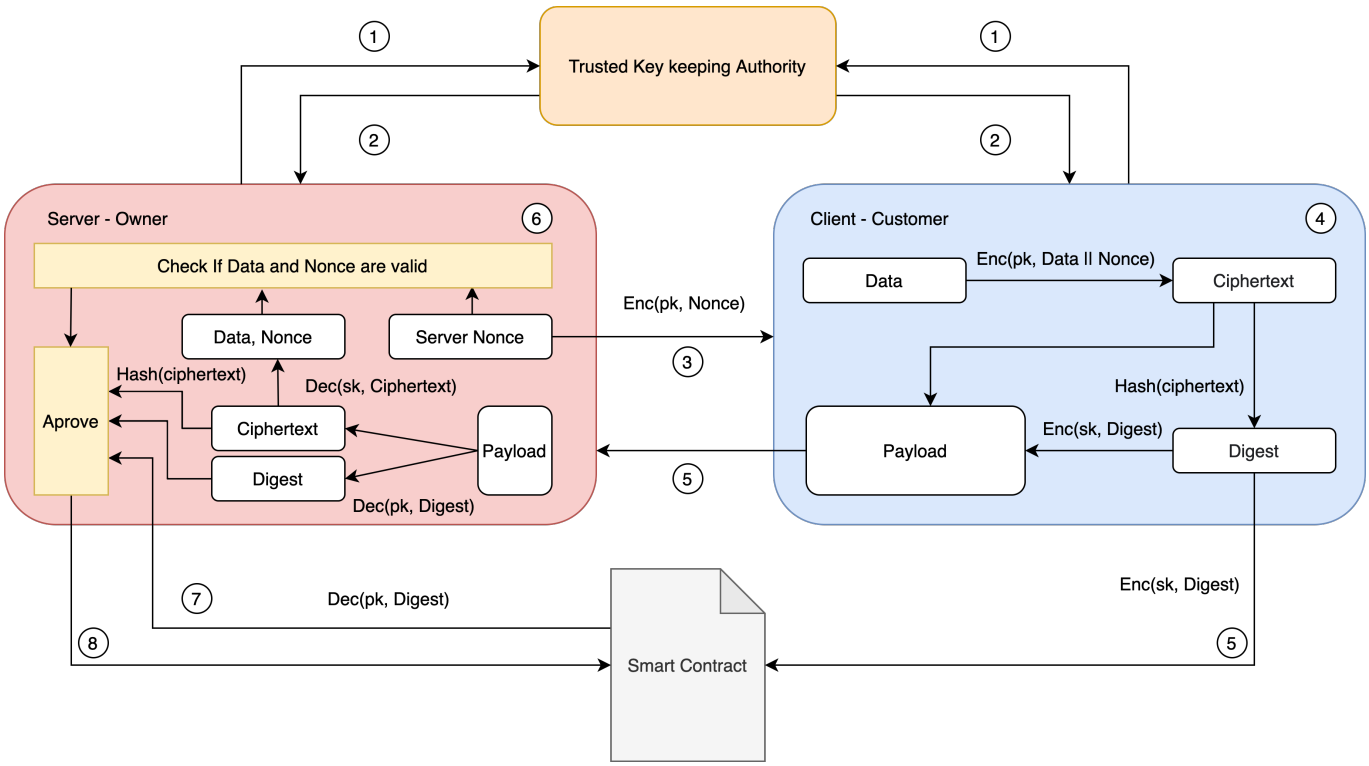

Figure 2: KYC process