BDL Assignment 3

Contract Design:

Internal variables:

For the internal purposes, five total internal (private) variables were used. These are listed as follows:

1. name – This is a string variable which stores the token name used in recognizing the token in wallets, scanners and other such usages. This can be retrieved by using the *getName* function.
2. symbol – This is a string variable which stores the token symbol used in recognizing the token in wallets, scanners and other such usages. This can be retrieved by using the *getSymbol* function.
3. price – This is an immutable uint128 type variable that stores the price value of the token. This can be retrieved by using the *getPrice* function.
4. _totalSupply – This is a uint type variable recording the total supply of the token. This can be retrieved by the *totalSupply* function.
5. balances – This is a map that stores balances of each address with the addresses of the users being key and the uint balances being the values.

Buying procedure:

To buy tokens, you would ideally (not mandatory as per code, as not defined by the contract requirements provided) send Ethereum comparable to the value of the desired number of tokens (price of token multiplied with number of tokens desired), plus some gas fees, to the contract which would be processed using the fallback function. The requirement for payment in lieu of tokens can be done off-chain, where the owner can confirm the transaction (as no public function can be implemented to check the same in the contract, as per requirement) and decide on the number of tokens to mint and send to the Ethereum sender in context. Only the owner of the contract can mint tokens, using the *mint* function, to send a certain value of tokens to a certain address. Upon a successful minting process, the number of tokens minted is added to both the balance of the receiving address, and the total supply variable _totalSupply. There is also a Mint event emitted by this contract registering the receiving address, and the number of tokens sent to the said address. This is the only way to “buy” tokens in this contract.

Transfer procedure:

The token may further be transferred between parties (addresses), if they have sufficient balances, using the *transfer* function. The sender must provide the address of the receiver and the number of tokens they wish to send. The balances of both the addresses are then adjusted accordingly, and a Transfer event is emitted, registering the sender’s address, receiver’s address and the number of tokens sent.

Selling procedure:

Finally, to sell tokens, the selling party needs to provide the number of tokens they wish to sell, using the *sell* function. Provided that they have sufficient balance in their address, the contract would remove that balance from their address in our *balances* map and send the token to the *burn address* (0) and subtract the same number from the total supply variable _totalSupply. Then the seller receives the value of the said tokens in wei (price of the tokens multiplied by the number of tokens) in their address. Upon successful completion of a sell, a Sell event is emitted registering the seller's address and the number of tokens sold.

Price of Token:

The price of token is pre-set during the invoking of the constructer function of the contract to 600 Wei per token, as given in the requirement specification for the contract. It can be accessed/verified using the *balanceOf* function.

Checking Balance:

To check the balance of an address, the function *balanceOf* is used. It takes as input, the address whose balance needs to be checked, and returns the said balance after retrieving it from the *balances* map.

The functions *getName*, *getPrice*, *getSymbol*, *totalSupply* and *balanceOf* can be used by wallets and scanners to retrieve the required data from the contract.

The *close* function:

This function can only be called by the owner of the contract, and it shuts down the contract, deletes the opcode, and send all the contract balance to the owner's address.

The *customLib* library:

This is a custom library deployed on the chain, that was implemented in the contract, and ultimately linked in the chain.

It has a *customSend* function that was implemented in the *send* function of this contract, which previously used *transfer* to send Wei to the seller's address in the part1 of the implementation. This method ensures that the sender is sending more than one token in the transfer. The first token is sent through a transfer call, and the rest of the tokens that need to be sent, are sent via *call* function.

This library is integrated by following the steps mentioned below:

1. Download source code and import into workspace in Remix.
2. Import the library into the contract using an import statement.
3. Use the method as required (demonstrated in the code section for part 2 of the appendix).
4. Compile the contract that is dependent on the library.
5. Open the eponymous (eponymous to the contract name) JSON file; *Token.json* in this case. Check under deploy section to choose which test net we need to use. It is "goerli:5", in this case. Update "<address>" to the contract address of the already deployed instance of the library source code in the respective test net; 0x9DA4c8B1918BA29eBA145Ee3616BCDFcFAA2FC51 in this case.
6. Change the value for the key *autoDeployLib* for that test net to false.
7. Deploy the contract that is dependent on the library in the required test net.

Thus, the contract is dynamically linked to the deployed instance of the library.

Gas Evaluation:

The view functions, namely *getName*, *getPrice*, *getSymbol*, *totalSupply* and *balanceOf* do not require gas fees.

For the specifications of the first part of the assignment, the following were the gas for the respective actions in the Goerli test net:

Action	Gas	Gas fees (GoerliETH)
Deploying	889964	0.002225
Sending Ether using CALL	21055	0.000053
Calling *mint* function	68892	0.000172
Calling *transfer* function	52336	0.000119
Calling *sell* function	64119	0.000135
Calling *close* function	26330	0.000066

To minimize the gas fees, the following steps were taken:

1. Minimizing reading and writing on-chain data: No variable that was not necessary for the functioning of the contract was declared or used. Number of functions were kept at minimal, only defining those functions required by the contract specification provided. Events were used instead of logs for registering transactions to further reduce memory usage.
2. Minimal usage of loops: No loops were used in this contract, and every function has a time-complexity of the form O(c), where c is a constant value.
3. Using more current versions of Solidity: Solidity 0.8.0+ versions are used for this contract, thus limiting the need for adding extra functions to do security checks; for instance – no safe-math function is required to check for overflow/underflow conditions.
4. Using immutable keyword: The address of the owner and the price were set to be immutable, as this helps us save gas.
5. Using view keyword: Setting functions to view functions prevent gas fees when calling them.
6. Processing the calculation of the number of tokens to be minted for an address off-chain: The requesting party sends the ether deemed necessary for purchasing tokens. The owner checks the transaction validity through the transaction ID or monitoring the chain and mints the respective number of tokens to send to the requesting party's address. This calculation of the number of tokens to be sent is done entirely off-chain to save gas fees.

To check the gas impact of using the deployed library instance compared to including its code in contract, both the versions were deployed and tested to yield the following results:

Using the external deployed library:

Action	Gas	Gas fees (GoerliETH)
Deploying	941981	0.002355
Calling *sell* function	77928	0.000194

Using library code included in the source code of the contract:

Action	Gas	Gas fees (GoerliETH)
Deploying	941969	0.002355
Calling *sell* function	77928	0.000194

As we see, the difference is not significant, but perhaps that is due to the size of the library. In fact, the higher gas for deployment of the contract that uses the externally linked pre-deployed library is probably because linking the contracts took more gas than it did for the internally implemented library deployment to process the extra code. It is expected that for large libraries, linking pre-deployed ones instead of reusing their code in the currently deploying contract would be cheaper.

Potential hazards and mitigation:

1. Denial of Service: This type of attack overloads the servicing capabilities of a contract, thus making it unavailable to the users, and hindering the functionality. To prevent these, the following mitigations are adopted:
    1. A pull mode of retrieval of balances is used, a push mode will not be practical for the requirements anyway.
    2. No unbounded loops used, or any loops for that matter.
2. Re-entrancy: A re-entrancy attack can happen when a function makes an external call to another untrusted contract. Then, in an attempt to drain tokens or coins, the untrusted contract makes a recursive call back to the original function. These can be mitigated using the Checks-Effects-Interactions pattern as follows:
    1. Checks are performed on the inputs, sender, values and arguments. In this use case, this is implemented in terms of overflow/underflow checks of solidity 0.8.0+, and authority level checking for certain functions which can only be run by the owner of the contract.
    2. Need to enforce effects and update the state accordingly. Thankfully, most of this for the required use case is handled natively by solidity, as any failing function or transaction simply reverts the contract's state.
    3. Interact with other accounts via external calls or send/transfer. Transfer is used for part 1, and external call (implementing transfer and CALL method) is used in part 2.
    4. Additionally, balance from a user's balance map is removed before transfer (mutex-like implementation) .
    5. payable.transfer() is used instead of call() in the native contract.
3. Preimage attack: If sensitive information is stored in a map, an attacker can game the system by figuring our preimages (keys) of information values in that map that are important to them. Thankfully, in this implementation, no such map is used to store sensitive information, so this is not a worry. The balances for addresses is public knowledge.

4. Unexpected Ether: In certain scenarios, ether can be forcibly sent to contracts in unexpected scenarios to game the system. Thankfully, in this case, there is no such exploit possible. Most importantly, this.balance is never used in this contract, as the only time the contract loses ether is when a token is sold, and since it is a fixed price system, and the issuer mints tokens only upon receival of funds, there is no need to track the balance of the token in the functions.
5. Delegate Call exploit: This happens when the attacker figures out a way to change the calling contract's data upon calling the library in a way that benefits them. To mitigate this, we simply never use calls of this format: <<library>>.delegatecall().
6. Tx.origin exploit: A transaction origin requirement can be exploited by an attacker by making a transaction reroute a call to a payment function in the original contract when it calls another contract (compromised contract). To prevent this, tx.origin is not used, and only msg.sender is used to verify source. Bounced calls are not considered.
7. Default visibilities exploit: A contract may have functions that should have been private, but no modifier has been mentioned, leading the function to fall back to the default public visibility. This can allow attackers to call such functions freely and hamper with the system. No extra functions were used in this contract, and the access control is verified in the required functions by verifying that the msg.sender matches the immutable owner's address.
8. Unchecked Call Return Values: For certain use cases in external calls, the CALL opcode can be used in Solidity. The call() and send() functions return a boolean to indicate if the call was successful. Thus they come with a simple caveat, in that the transaction that executes these functions will not revert if the external call triggered by call() or send() fails. The call() or send() will simply return false. When the return value is not checked, the system enters an erroneous state due to the absence of a revert. In our case, the Boolean is in fact checked in the implementation for part 2, and if the transaction indeed fails, the contract state will revert back.
9. Block Timestamp Manipulation: Block timestamps when used as a source of randomness can become a huge vulnerability, when or if the attacker can win the next block(s), thus determining their timestamp, and controlling the randomness into their favour. This is normally mediated by not using block timestamps, or using the 15 seconds rule. Neither is required for this use case, as we do not use block timestamps anyway.
10. Default value exploit (e.g., Nomad-Bridge Hack) and control flow disruption: Default values are initialised properly.
11. Front-running: Frontrunning is an issue where the attacker tries to perform the same action as the victim, but does it sooner, usually, by outbidding the victim's gas. Since in this case, copying or using time-sensitive information is not a problem, the only thing to do is to remove the benefit of front-running. This is done by making the price stay at a constant 600 Wei per token. So, if a party is trying to sell their token faster, they will just spend more gas. No particular exploit will come out of it.
12. Overflow/Underflow exploit: Solidity version >=0.8.0 is used, which has an overflow/underflow check, and will revert contract state if any transaction or non-view function call fails.
13. Pseudo-randomness: The pseudo-randomness schemes used in smart contracts can in fact be predictable in certain cases. If the attacker can figure out the pattern in

which the pseudo-randomness is generated, they can exploit the system in their favour. In this use case, no such randomness is used, so no mediation is necessary.

14. Rug pull exploit: This is when the owner can remove the liquidity of a token by removing the contract balance in entirety. As per the contract specifications provided, the *close* function does exactly that. This is a gravely serious security risk, and the only way to mitigate this is by removing the said function and making sure that no one can remove the liquidity of the token as and when they choose to.

KYC:

A public-key encryption scheme uses a pair of keys – a public key and a private key. The public key is known to every party on a network, and the private key is only kept with the individual to whom the set of keys were assigned, who needs to prove their identity. Any messages or documents in such systems are encrypted by the public key before putting it on the system and the private key is used to decipher the same.

In the given use case, upon receiving and verifying the document off-chain, a pair of public and private keys could be generated at the contract issuer's end. The document is to be encrypted using the public key. The encrypted document could be stored on-chain, although that would be costly. Off-chain economic storage is advised. The private key is stored using some robust Key Management System (which indexes the private key using the public key) off-chain. The public key and the pointer (any reference system identifier) to the document can be stored via passing it as an argument into the *Mint* event. So, both the *mint* function and the *Mint* event would have the two added parameters: the public key, and the pointer to the encrypted document. For any future purposes, when the document needs to be retrieved, one can refer to the event log using the transaction ID for the *mint* function call and obtain the public key and document pointer pair. Upon accessing the encrypted document securely using the document pointer, the document can be decrypted using the private key retrieved from the KMS using the public key, off-chain. If we want to do the whole process on-chain, both the document and the private key would be revealed to the network at some point, which is less than undesirable. So, off-chain processing of the file is necessary.

Appendix:

Transaction History:

Part 1:

ID type	ID
Deployment transaction ID	0x9f216bc4a2f27a413552cefd6aaf14d272f1f95afd5a37933555e555d4e9291b
contract address	0x21cFda5b23F88a49076Adc41D0A13fDEB2d4324d
Transaction ID for loading Eth:	0x00d2b1c10f3b048b5b3bb456cb836b76d2fe2ab8872d359c460dd04aaed18db8
Address of owner	0x181c1235a7029f5b095C067732952fA33Fedd9b8
Transaction ID of mint operation	0x60419d0c0eff5e3dd023c0a654a4e902ae59d0cd0c5d005cd4909b607d49db71

Transaction ID of transfer operation	0xc31105340d14837d7be182c561be7ecc3386a3ed389a1efb201dbc91aaf7bee2
Transaction ID of sell operation	0x86a988c03562c14f01219c17015588c19257c6a9cd95a01a2508c130898d550d
Transaction ID of close operation	0x5936082c06c8dcc68f4e553692f839e218807c5a011eeea630c5554a269cb7ee

Part 2:

ID type	ID
Deployment transaction ID	0xbc1f607e0af7e5a6fd17979d7690ca52fef23d888638824c550bf793f03d9089
contract address	0x2C991beD38903a3a8127adE662686aDfD5046c25
Address of owner	0xA06E1629d640230167678ddBfD817E2ad4E706A8

Code:

Part 1:

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

pragma solidity >=0.8.0 <0.9.0;

contract Token {
    address payable public immutable owner;

    string private name;
    string private symbol;
    uint128 private immutable price;
    uint private _totalSupply;

    mapping(address => uint) private balances;

    event Transfer(address indexed_from, address indexed_to, uint256 value);
    event Mint(address indexed_to, uint256 value);
    event Sell(address indexed_from, uint256 value);

    constructor() {
        owner = payable(msg.sender);
        name = "Test Token";
        symbol = "TEST_TOKEN";
        price = 600;
        _totalSupply = 0;
    }

    function totalSupply() public view returns (uint256){
        return _totalSupply;
```

```
    }

    function balanceOf(address _account) public view returns (uint256){
        return balances[_account];
    }

    function getName() public view returns (string memory){
        return name;
    }

    function getSymbol() public view returns (string memory){
        return symbol;
    }

    function getPrice() public view returns (uint128){
        return price;
    }

    function transfer(address to, uint256 value) public returns (bool){
        balances[msg.sender] =
            balances[msg.sender] - value;
        balances[to] =
            balances[to] + value;
        emit Transfer(msg.sender, to, value);
        return true;

    }

    function mint(address to, uint256 value) public returns (bool){
        require(payable(msg.sender) == owner);
        _totalSupply = _totalSupply + value;
        balances[to] =
            balances[to] + value;
        emit Mint(to, value);
        return true;
    }

    function sell (uint256 value) public payable returns (bool){
        balances[msg.sender] =  balances[msg.sender] - value;
        balances[address(0)] = balances[address(0)] + value;
        _totalSupply = _totalSupply - value;
        payable(msg.sender).transfer(price * value);
        emit Sell (msg.sender, value);
        return true;
    }

    function close () public {
        require (payable(msg.sender) == owner);
        selfdestruct(payable(msg.sender));
    }
```

```
    receive() external payable {
    }

}
```

Part 2:

```
// SPDX-License-Identifier: GPL-3.0
import "Assignment 3/customLib.sol";

pragma solidity >=0.8.0 <0.9.0;

contract Token {
    address payable public immutable owner;

    string private name;
    string private symbol;
    uint128 private immutable price;
    uint private _totalSupply;

    mapping(address => uint) private balances;

    event Transfer(address indexed_from, address indexed_to, uint256 value);
    event Mint(address indexed_to, uint256 value);
    event Sell(address indexed_from, uint256 value);

    constructor() {
        owner = payable(msg.sender);
        name = "Test Token";
        symbol = "TEST_TOKEN";
        price = 600;
        _totalSupply = 0;
    }

    function totalSupply() public view returns (uint256){
        return _totalSupply;
    }

    function balanceOf(address _account) public view returns (uint256){
        return balances[_account];
    }

    function getName() public view returns (string memory){
        return name;
    }

    function getSymbol() public view returns (string memory){
        return symbol;
    }
```

```
    function getPrice() public view returns (uint128){
        return price;
    }

    function transfer(address to, uint256 value) public returns (bool){
        balances[msg.sender] =
            balances[msg.sender] - value;
        balances[to] =
            balances[to] + value;
        emit Transfer(msg.sender, to, value);
        return true;
    }

    function mint(address to, uint256 value) public returns (bool){
        require(payable(msg.sender) == owner);
        _totalSupply = _totalSupply + value;
        balances[to] =
            balances[to] + value;
        emit Mint(to, value);
        return true;
    }

    function sell (uint256 value) public payable returns (bool){
        balances[msg.sender] =  balances[msg.sender] - value;
        balances[address(0)] = balances[address(0)] + value;
        _totalSupply = _totalSupply - value;
        customLib.customSend(price * value, msg.sender);
        emit Sell (msg.sender, value);
        return true;
    }

    function close () public {
        require (payable(msg.sender) == owner);
        selfdestruct(payable(msg.sender));
    }

    receive() external payable {
    }

}
```