# SMART CONTRACT AUDIT REPORT

for

# SmartDeFi

Prepared By: Xiaomi Huang

PeckShield

February 10, 2023

## Document Properties

| | |
|---|---|
| Client | FEG |
| Title | Smart Contract Audit Report |
| Target | SmartDeFi |
| Version | 1.1 |
| Author | Luck Hu |
| Auditors | Luck Hu, Xuxian Jiang |
| Reviewed by | Patrick Lou |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author | Description |
|---|---|---|---|
| 1.1 | February 10, 2023 | Luck Hu | Post Release #1 |
| 1.0 | February 2, 2023 | Luck Hu | Final Release |
| 1.0-rc | December 6, 2022 | Luck Hu | Release Candidate |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.



| | |
|---|---|
| Name | Xiaomi Huang |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related source code of `SmartDeFi`, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to either security or performance. This document outlines our audit results.

## 1.1 About SmartDeFi

`SMARTDeFi` token is an `ERC20`-compliant token, which is designed to be a reflection token. It is based on the `SmartDeFi` technology which is backed by a separate locked pool of funds that will consistently grow with each transaction. The basic information of the audited `SmartDeFi` is as follows:

Table 1.1: Basic Information of SmartDeFi

| Item | Description |
|---:|---|
| Name | FEG |
| Type | Ethereum ERC20 Token Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Audit Completion Date | February 10, 2023 |

In the following, we show the git repository and the commit hash value used in this audit. Note the audit scope only covers the implemented contracts in file rebuilt fee structure.sol, and those contracts (e.g., `DATA_READ`) that are used but not implemented in the file are not in the audit scope.

- https://github.com/FEG-team/SD-2.0 (2772652a)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/FEG-team/SD-2.0 (4d326da0)

## 1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

| **Impact** \ **Likelihood** | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

| Category | Check Item |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead of Transfer |
| | Costly Loop |
| | (Unsafe) Use of Untrusted Libraries |
| | (Unsafe) Use of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| | Approve / TransferFrom Race Condition |
| **ERC20 Compliance Checks** | Compliance Checks (Section 3) |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the `SmartDeFi` contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 3 | ■ ■ ■ |
| Low | 1 | ■ |
| Informational | 0 | |
| Total | 4 | |

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

## 2.2 Key Findings

Overall, a minor ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key SmartDeFi Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Suggested Access Control to Set onlySB | Coding Practices | Fixed |
| PVE-002 | Low | Accommodation of Non-ERC20-Compliant Tokens | Coding Practices | Fixed |
| PVE-003 | Medium | Possible Sandwich/MEV Attacks in SmartDeFi | Time and State | Partly Fixed |
| PVE-004 | Medium | Trust Issue of Admin Keys | Security Features | Confirmed |

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 4 for details.

# 3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

| Item | Description | Status |
|---|---|---|
| **name()** | Is declared as a public view function | ✓ |
| | Returns a string, for example "Tether USD" | ✓ |
| **symbol()** | Is declared as a public view function | ✓ |
| | Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length | ✓ |
| **decimals()** | Is declared as a public view function | ✓ |
| | Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required | ✓ |
| **totalSupply()** | Is declared as a public view function | ✓ |
| | Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment | ✓ |
| **balanceOf()** | Is declared as a public view function | ✓ |
| | Anyone can query any address' balance, as all data on the blockchain is public | ✓ |
| **allowance()** | Is declared as a public view function | ✓ |
| | Returns the amount which the spender is still allowed to withdraw from the owner | ✓ |

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue found in the audited SmartDeFi. Specifically, the `Transfer()` event is not emitted for zero value transfer when the sender/recipient are not whitelisted.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

| Item | Description | Status |
|---|---|---|
| **transfer()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the caller does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | — |
| **transferFrom()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the spender does not have enough token allowances to spend | ✓ |
| | Updates the spender's token allowances when tokens are transferred successfully | ✓ |
| | Reverts if the from address does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | — |
| **approve()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token approval status | ✓ |
| | Emits Approval() event when tokens are approved successfully | ✓ |
| **Transfer()** event | Is emitted when tokens are transferred, including zero value transfers | — |
| | Is emitted with the from address set to $address(0x0)$ when new tokens are generated | ✓ |
| **Approval()** event | Is emitted on any successful call to approve() | ✓ |

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

| Feature | Description | Opt-in |
|---|---|---|
| **Deflationary** | Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls | ✓ |
| **Rebasing** | The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address | — |
| **Pausable** | The token contract allows the owner or privileged users to pause the token transfers and other operations | ✓ |
| **Blacklistable** | The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited | — |
| **Mintable** | The token contract allows the owner or privileged users to mint tokens to a specific address | — |
| **Burnable** | The token contract allows the owner or privileged users to burn tokens of a specific address | ✓ |

# 4 Detailed Results

## 4.1 Suggested Access Control to Set onlySB

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SMARTDeFi
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

In the SMARTDeFi contract, there is a state variable, i.e., onlySB, which controls how the transaction is taxed. Per design, when the onlySB is true, it only taxes on buy/sell transactions. When the onlySB is false, it taxes on all transactions. While examining the update of the onlySB logic, we notice there is no access control for the afterLP() routine, which allows anyone to reset the onlySB back to the initial value.

To elaborate, we show below the code snippet of the afterLP() routine. As the name indicates, it is used to be called after the liquidity has been added to enable the exchange taxes. It resets the onlySB to preLiqSB which is the default value initialized in the constructor. However, there is a lack of proper access control in this routine and anyone can call it at any time. As a result, the transactions may be taxed unexpectedly.

```
    function afterLP() external{
        isExchange[uniswapV2Pair] = true;
        isExchange[secondV2Pair] = true;
        onlySB = preLiqSB;
    }
```

Listing 4.1: SMARTDeFi::afterLP()

**Recommendation** Add necessary access control in the afterLP() routine.

**Status** The issue has been fixed by the following commit: 562a2a51.

## 4.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SafeTransfer
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 4.2: USDT::**transfer**()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `_pushUnderlying()` routine in the `SafeTransfer` library. If the `USDT` token is supported as `erc20`, the unsafe version of `IERC20(erc20).transfer(to, amount)` (line 202) returns no value. Hence the following validation of the return value fails (line 203), and the transaction reverts.

```
200     function _pushUnderlying(address erc20, address to, uint amount) internal
201     {
202         bool xfer = IERC20(erc20).transfer(to, amount);
203         require(xfer,"Push");
204     }
```

Listing 4.3: SafeTransfer :: _pushUnderlying()

**Recommendation** Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

**Status** The issue has been fixed by the following commit: `562a2a51`.

## 4.3 Potential Sandwich/MEV Attack in SmartDeFi

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

### Description

The `SMARTDeFi` token is designed to be deflationary, which means it charges fees on token transfer. Specifically, it charges fees like backing fee, burning fee and liquidity fee, etc. In particular, the backing fee will be converted to the backing token which will be locked in the contract as an inherent value for the investors. And the liquidity fee will be supplied to the dedicated pool in `UniswapV2` to incentivize the LPs.

To elaborate, we show below the code snippets from the `BackingLogic` contract. As the name indicates, the `convertBacking()` routine is used to convert the accumulated backing fee to the backing asset. The exchange is fulfilled by the `UNISWAP_V2_ROUTER`, where the `amountOutMin` is set to 0 (line 435), which means it does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks. As a result, a bad actor can swap its `SMARTDeFi` tokens to the backing asset to raise the backing asset price before the `convertBacking()` transaction, and does reverse swap afterward to make a profit.

Similarly, the `convertLiquidity()` routine is used to supply the accumulated liquidity fee into the dedicated pool in `UniswapV2`. It converts half of the liquidity fee to `WETH` first (line 450), which is supplied together with the other half of the liquidity fee to the pool in `UniswapV2` as new liquidity (line 452). In the `swapTokensForEth()` routine which is used to convert the liquidity fee to `WETH`, the `amountOutMin` is also set to 0 (line 464), which means it does not specify any restriction on possible slippage either, and is therefore vulnerable to possible front-running attacks. Based on this, it is suggested to add proper slippage control in both `convertBacking()/swapTokensForEth()` routines.

What is more, the `addLiquidity1()` routine is used to add half of the liquidity fee and the converted `WETH` to `UniswapV2` as new liquidity. It directly transfers both tokens into the pool and invokes the `sync()` routine to apply the new liquidity. However, this is also vulnerable to possible front-running attacks, where a bad actor can add liquidity to mint `LP` tokens before the `addLiquidity1()` transaction, and remove the liquidity afterward to make a profit. Note the same issue is also applicable to the `SMARTDeFi::manualBurn()` routine.

```
426     function convertBacking(uint liqShare, uint backingThreshold) private {
427         uint256 amt = IERC20(mainToken).balanceOf(address(this)) - (collateral +
                liqShare);
428         if (amt >= backingThreshold) {
429             IERC20(mainToken).approve(address(UNISWAP_V2_ROUTER), amt);
430             address[] memory path;
431             if(backingAsset == WETH){...}
432             else{...}
433             IUniswapV2Router02(UNISWAP_V2_ROUTER).
                    swapExactTokensForTokensSupportingFeeOnTransferTokens(
434                 amt,
435                 0,
436                 path,
437                 address(this),
438                 block.timestamp
439             );
440         }
441     }
442
443     function convertLiquidity(uint liqShare, uint liquidityThreshold) private returns(
            uint) {
444         uint256 amt = liqShare;
445         if (amt >= liquidityThreshold) {
446             uint256 half = amt.div(2);
447             uint256 otherHalf = amt.sub(half);
448             uint256 initialBalance = IERC20(WETH).balanceOf(address(this));
449             IERC20(mainToken).approve(address(UNISWAP_V2_ROUTER), amt);
450             swapTokensForEth(half,address(this));
451             uint256 newBalance = IERC20(WETH).balanceOf(address(this)).sub(
                    initialBalance);
452             addLiquidity1(otherHalf, newBalance);
453             return 0;
454             }
455             return liqShare;
456     }
457
458     function swapTokensForEth(uint256 tokenAmount,address toSend) private {
459         address[] memory path = new address[](2);
```

```
        path[0] = mainToken;
        path[1] = WETH;
        IUniswapV2Router02(UNISWAP_V2_ROUTER).swapExactTokensForTokensSupportingFeeOnTransferTokens(
            tokenAmount,
            0,
            path,
            toSend,
            block.timestamp
        );
    }

    function addLiquidity1(uint256 tokenAmount, uint256 ethAmount) private {
        IERC20(mainToken).transfer(mainPair,tokenAmount);
        IERC20(WETH).transfer(mainPair,ethAmount);
        IPair(mainPair).sync();
    }
```

Listing 4.4: `BackingLogic` contract

**Recommendation** Improve the above mentioned routines by adding necessary slippage control to protect the protocol from potential Sandwich/MEV attack.

**Status** This issue in the `SMARTDeFi::manualBurn()` routine has been fixed by adding a slippage control. Regarding other above-mentioned routines, the issue has been confirmed and partly mitigated by the project team. The mitigation is achieved by applying a limitation to the `SMARTDeFi` token transfer, which requires the transaction sender (`tx.origin`) must be the transfer sender or recipient. With this limitation, token transfers between two contracts are forbidden. As a result, this could block such attack pattern that the attacker triggers the attack via a delegation contract and the `SMARTDeFi` tokens are to be transfered between the attack contract and the victim protocol. However, this mitigation does not have effect on other attack patterns if the attacker does not use a delegation contract or the `SMARTDeFi` tokens are transfered between the attacker and the victim protocol directly.

## 4.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SMARTDeFi
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In the SmartDeFi protocol, there are certain privileged accounts, i.e., owner/admin, that play critical roles in governing and regulating the system-wide operations (e.g., set the fees rates). Our analysis shows that the privileged accounts need to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the owner/admin accounts.

Firstly, the owner holds all the token supply of SMARTDeFi when the contract is deployed (line 964), which means the owner has the right to distribute the tokens.

```
945     constructor(string memory n, string memory symbol_, uint supply, uint[12] memory fee
            , bool sb,address ownerAddr, address backingTokenAddress) {
946         require(IERC20(backingTokenAddress).balanceOf(address(this))==0,"invalid backing
                ");
947         IUniswapV2Router02 _uniswapV2Router = IUniswapV2Router02(UNISWAP_V2_ROUTER);
948         uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory())
949         .createPair(address(this), WETH);
950         secondV2Pair = IUniswapV2Factory(IUniswapV2Router02(SECOND_V2_ROUTER).factory())
951         .createPair(address(this), WETH);
952         _tTotal = supply * 10 ** 18;
953         require(_tTotal <= type(uint128).max);
954         //required because of pcs limitation
955         _rTotal = (MAX - (MAX % _tTotal));
956         backingThreshold = _tTotal / 10000;
957         liquidityThreshold = _tTotal / 10000;
958         name = n;
959         symbol = symbol_;
960         sdOwner = ownerAddr;
961         reflection = fee[5] % 1000;
962         allFee = fee;
963         sdFeeRecipient = ownerAddr;
964         _rBank[ownerAddr] = _rTotal;
965         emit Transfer(address(0), ownerAddr, _tTotal);
966         ...
967     }
```

Listing 4.5: SMARTDeFi::constructor()

Secondly, the privileged functions in the SMARTDeFi contract allow for the owner/admin to set the fees (up to 50% of the transfer amount) and add/remove tax free users. The fees are taken from the sender of a token transfer. Note no fee is taken from a token transfer if the sender or the recipient are tax-free user.

```
    function setTaxFreeUser(address user, bool adding) external {
        require(msg.sender == sdOwner);
        taxFree[user] = adding;
    }

    function suggestSetFee(uint[12] calldata feeSet, bool onlySellBuy) external {
        bool admin = Reader(DATA_READ).isAdmin(msg.sender);
        require(msg.sender == sdOwner||admin);
        Verifier.check(feeSet);
        suggestedAllFee = feeSet;
        suggestedOnlySB = onlySellBuy;
        timeDelay = admin ? block.timestamp : block.timestamp + 3 days; // 3 days grace
            periode
    }
```

Listing 4.6: Example Privileged Operations in the `SMARTDeFi` Contract

Thirdly, the privileged functions in the `BackingLogic` contract allow for the `admin` to set the farming address which is used to receive the new LPs that are minted from the liquidity fees. The farming is designed to incentivize the staking of the supported LPs with the new received LPs.

```
    function addFarmingAddress(address farmingAddress) external{
        require(Reader(DATA_READ).isAdmin(msg.sender));
        farmAddress=farmingAddress;
        farmEnable = (farmingAddress != address(0));
    }
```

Listing 4.7: Example Privileged Operations in the `BackingLogic` Contract

Lastly, the privileged functions in the `SMARTDeFi` contract allow for the `owner/admin` to add/remove exchanges, set the fee recipient, and set the staking address, etc. Note a token transfer to an exchange is recognized as a buy action, which will be taken buy fees. All other token transfers are taken as sell actions which will be taken sell fees. The fee recipient is used to receive the protocol fees, and the staking address is used to receive the staking fees.

```
    function setExchange(address LP,bool adding) external{
        require(msg.sender == sdOwner);
        isExchange[LP] = adding;
    }

    function setSDFeeRecipient(address addy) external {
        require(addy != address(0));
        require(msg.sender == sdOwner);
        sdFeeRecipient = addy;
    }

    function setStakingAddress(address addy) external {
        require(msg.sender == Reader(DATA_READ).stakeDeployerAddress());
```

```
        sdStake = addy;
    }
```

Listing 4.8: Example Privileged Operations in the SMARTDeFi Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner/admin may also be a counter-party risk to the protocol users. It is worrisome if the privileged owner/admin accounts are plain EOA accounts. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

**Recommendation** Promptly transfer the privileged accounts to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** This issue has been confirmed by the project team and the token creator can decide how to manage the privileged accounts.

# 5 Conclusion

In this security audit, we have examined the design and implementation of `SmartDeFi`. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified four issues of varying severities. And the identified issues have been fixed, confirmed or mitigated. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

# References


[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.