



SMART CONTRACT AUDIT REPORT

for

FilDA Protocol



Prepared By: Yiqun Chen

PeckShield
December 2, 2021

Document Properties

Client	FiIDA
Title	Smart Contract Audit Report
Target	FiIDA
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 2, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 29, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About FilDA	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Uninitialized State Index DoS From Reward Activation	12
3.2	Improved Logic of QsSushiLPDelegate::doTransferIn()	15
3.3	Proper stSushiPerShare Accounting in WMiniChefV2	16
3.4	Accommodation of Non-ERC20-Compliant Tokens	18
3.5	Trust Issue of Admin Keys	20
4	Conclusion	22
	References	23

1 | Introduction

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

1.1 About FiLDA

FiLDA is a lending and borrowing protocol with the goal of developing a cross-chain money market. The protocol designs are architected and inspired based on Compound and Alpha HomoraV2 and synced into the FiLDA platform to capitalize the benefits of both systems. FiLDA enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by staking over-collateralized cryptocurrencies. It has been launched on HECO and will be deployed on Ethereum and other public chains.

The basic information of FiLDA is as follows:

Table 1.1: Basic Information of FiLDA

Item	Description
Name	FiLDA
Website	https://www.filda.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 2, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- <https://github.com/fildaio/compound-protocol.git> (d8a2351)
- <https://github.com/fildaio/alpha-homora-v2-contract.git> (7d1686e)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- <https://github.com/fildaio/compound-protocol.git> (a81306d)
- <https://github.com/fildaio/alpha-homora-v2-contract.git> (4faa6f5)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems 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).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis 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.

In particular, we perform the audit according to the following procedure:

- 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.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
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

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.




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 implementation of the `FILDA` protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	
Medium	1	
Low	2	
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key FilDA Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Fixed
PVE-002	Low	Improved Logic of QsSushiLPDelegate::doTransferIn()	Business Logic	Fixed
PVE-003	High	Proper stSushiPerShare Accounting in WMiniChefV2	Business Logic	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Feature	Mitigated

Besides 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 3 for details.

3 | Detailed Results

3.1 Uninitialized State Index DoS From Reward Activation

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Qstroller
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `FiLDA` protocol provides incentive mechanisms that reward the protocol users. Specifically, the reward mechanism follows the same approach as the `COMP` reward in `Compound`. Our analysis on the related `COMP` reward in `FiLDA` shows the current logic needs to be improved.

To elaborate, we show below the initial logic of `_setCompSpeeds()` that kicks off the actual minting of protocol tokens. It comes to our attention that the initial supply-side index is configured on the conditions of `compSupplyState[cToken].index == 0` and `compSupplyState[cToken].block == 0` (line 908). However, for an already listed market with a current speed of 0, the first condition is indeed met while the second condition does not! The reason is that both supply-side state and borrow-side state have the associated block information updated, which is diligently performed via other helper pairs `updateCompSupplyIndex()/updateCompBorrowIndex()`. As a result, the `_setCompSpeedInternal()` logic does not properly set up the default supply-side index and the default borrow-side index.

```

49     function _setCompSpeeds(address[] memory _allMarkets, uint[] memory _compSpeeds)
        public {
50         // Check caller is admin
51         require(msg.sender == admin);

53         require(_allMarkets.length == _compSpeeds.length);

55         for (uint i = 0; i < _allMarkets.length; i++) {
56             _setCompSpeedInternal(_allMarkets[i], _compSpeeds[i]);
57         }
58     }

```

```

60     function _setCompSpeedInternal(address _cToken, uint _compSpeed) internal {
61         Market storage market = markets[_cToken];
62         if (market.isComped == false) {
63             _addCompMarketInternal(_cToken);
64         }
65         uint currentCompSpeed = compSpeeds[_cToken];
66         uint currentSupplySpeed = currentCompSpeed >> 128;
67         uint currentBorrowSpeed = uint128(currentCompSpeed);

69         uint newSupplySpeed = _compSpeed >> 128;
70         uint newBorrowSpeed = uint128(_compSpeed);
71         if (currentSupplySpeed != newSupplySpeed) {
72             updateCompSupplyIndex(_cToken);
73         }
74         if (currentBorrowSpeed != newBorrowSpeed) {
75             Exp memory borrowIndex = Exp({mantissa: CToken(_cToken).borrowIndex()});
76             updateCompBorrowIndex(_cToken, borrowIndex);
77         }
78         compSpeeds[_cToken] = _compSpeed;
79     }

```

Listing 3.1: Qstroller::_setCompSpeeds()

```

900     function _addCompMarketInternal(address cToken) internal {
901         Market storage market = markets[cToken];
902         require(market.isListed == true, "!listed");
903         require(market.isComped == false, "already added");

905         market.isComped = true;
906         emit MarketComped(CToken(cToken), true);

908         if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
909             compSupplyState[cToken] = CompMarketState({
910                 index: compInitialIndex,
911                 block: safe32(getBlockNumber(), "exceeds 32 bits")
912             });
913         }

915         if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
916             compBorrowState[cToken] = CompMarketState({
917                 index: compInitialIndex,
918                 block: safe32(getBlockNumber(), "exceeds 32 bits")
919             });
920         }
921     }

```

Listing 3.2: Comptroller::_addCompMarketInternal()

When the reward speed is configured, since the supply-side and borrow-side state indexes are not initialized, any normal functionality such as `mint()` will be immediately reverted! This revert occurs inside the `distributeSupplierComp()/distributeBorrowerComp()` functions. Using the `distributeSupplierComp`

() function as an example, the revert is caused from the arithmetic operation `sub_(supplyIndex, supplierIndex)` (line 820). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateCompSupplyIndex()` (line 72). However, when the `distributeSupplierComp()` function is invoked, the `supplierIndex` is reset with `compInitialIndex` (line 817), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```

810     function distributeSupplierComp(address cToken, address supplier, bool distributeAll
      ) internal {
811         CompMarketState storage supplyState = compSupplyState[cToken];
812         Double memory supplyIndex = Double({mantissa: supplyState.index});
813         Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][
            supplier]});
814         compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

816         if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
817             supplierIndex.mantissa = compInitialIndex;
818         }

820         Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
821         uint supplierTokens = CToken(cToken).balanceOf(supplier);
822         uint supplierDelta = mul_(supplierTokens, deltaIndex);
823         uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
824         compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ?
            0 : compClaimThreshold);
825         emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta,
            supplyIndex.mantissa);
826     }

```

Listing 3.3: `Comptroller::distributeSupplierComp()`

Recommendation Properly initialize the reward state indexes in the above affected `_addCompMarketInternal` () function. An example revision is shown as follows:

```

900     function _addCompMarketInternal(address cToken) internal {
901         Market storage market = markets[cToken];
902         require(market.isListed == true, "!listed");
903         require(market.isComped == false, "already added");

905         market.isComped = true;
906         emit MarketComped(CToken(cToken), true);

908         if (compSupplyState[cToken].index == 0) {
909             compSupplyState[cToken].index = compInitialIndex;
910         }
911         compSupplyState[cToken].block = safe32(getBlockNumber());

913         if (compBorrowState[cToken].index == 0) {
914             compBorrowState[cToken].index = compInitialIndex;
915         }
916         compBorrowState[cToken].block = safe32(getBlockNumber());

```

917

}

Listing 3.4: `Comptroller::_addCompMarketInternal()`

Status The issue has been fixed by this commit: 920d8a0.

3.2 Improved Logic of `QsSushiLPDelegate::doTransferIn()`

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `QsSushiLPDelegate`, `QsSushiLPDelegate`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `FilDA` protocol has the lending pool component, which is in essence an over-collateralized lending protocol that supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. In the following, we examine the support of wrapping `SushiLP` as collateral.

To elaborate, we show below the related `doTransferIn()` function, which is designed to transfer in the underlying assets and sweep into `sushiPool`. While it properly deposits the underlying asset into `sushiPool`, it misses the call to timely claim possible `SUSHI` and `COMP` rewards (via `claimRewardsFromSushi()`).

```

246  /**
247   * @notice Transfer the underlying to this contract and sweep into master chef
248   * @param from Address to transfer funds from
249   * @param amount Amount of underlying to transfer
250   * @return The actual amount that is transferred
251   */
252  function doTransferIn(address from, uint amount) internal returns (uint) {
253      // Perform the EIP-20 transfer in
254      EIP20Interface token = EIP20Interface(underlying);
255      require(token.transferFrom(from, address(this), amount), "!transfer");
256
257      // Deposit to sushi pool.
258      sushiPool.deposit(pid, amount, address(this));
259
260      updateLPSupplyIndex();
261      updateSupplierIndex(from);
262
263      mintToFilda();
264  }

```

```

265     return amount;
266 }

```

Listing 3.5: QsSushiLPDelegate::doTransferIn()

In addition, we notice the need of calling `mintToFilda()` after all indexes are updated (lines 260-263). However, this is violated in the `seizeInternal()` routine from the same contract. Note that the same issue is also applicable to two other contracts `QsQuickLPDelegate` and `QsQuickDualLPDelegate`.

```

281     function seizeInternal(address seizerToken, address liquidator, address borrower,
282         uint seizeTokens) internal returns (uint) {
283         claimRewardsFromSushi();
284         updateLPSupplyIndex();
285         updateSupplierIndex(liquidator);
286         updateSupplierIndex(borrower);
287
288         mintToFilda();
289
290         address safetyVault = Qstroller(address(comptroller)).qsConfig().safetyVault();
291         updateSupplierIndex(safetyVault);
292
293         return super.seizeInternal(seizerToken, liquidator, borrower, seizeTokens);
294     }

```

Listing 3.6: QsSushiLPDelegate::seizeInternal()

Recommendation Timely claim rewards so that they can fairly distributed to current protocol users.

Status The issue has been fixed by this commit: [f848a69](#).

3.3 Proper stSushiPerShare Accounting in WMiniChefV2

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: WMiniChefV2, WStakingDualRewards
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `FILDA` protocol has another farming component that extends the solid base of `Alpha HomoraV2` to further wrap additional staking contracts including `WMiniChefV2` and `WStakingDualRewards`. While examining the extended support, we notice the pairing logic needs to be improved.

In particular, we show below the related mint/burn pair from the WMiniChefV2 contract. We notice that the accounting information of sushiPerShare is properly maintained for each ERC1155-based token id. However, another accounting information of stRewardPerShare[pid] is saved per each pool id. In other words, the stRewardPerShare[pid] state is not specific for each ERC1155-based token id, which unfortunately leads to improper reward distribution when the token is burnt.

```

62  function mint(uint pid, uint amount) external nonReentrant returns (uint) {
63      address lpToken = chef.lpToken(pid);
64      IERC20(lpToken).safeTransferFrom(msg.sender, address(this), amount);
65      if (IERC20(lpToken).allowance(address(this), address(chef)) != uint(-1)) {
66          // We only need to do this once per pool, as LP token's allowance won't decrease
67          // if it's -1.
68          IERC20(lpToken).safeApprove(address(chef), uint(-1));
69      }
70      chef.deposit(pid, amount, address(this));
71
72      IRewarder rewarder = chef.rewarder(pid);
73      (stRewardPerShare[pid], ) = rewarder.poolInfo(pid);
74
75      (uint128 sushiPerShare, ) = chef.poolInfo(pid);
76      uint id = encodeId(pid, sushiPerShare);
77      _mint(msg.sender, id, amount, '');
78      return id;
79  }

```

Listing 3.7: WMiniChefV2::mint()

```

85  function burn(uint id, uint amount) external nonReentrant returns (uint) {
86      if (amount == uint(-1)) {
87          amount = balanceOf(msg.sender, id);
88      }
89      (uint pid, uint stSushiPerShare) = decodeId(id);
90      _burn(msg.sender, id, amount);
91      chef.withdrawAndHarvest(pid, amount, address(this));
92      address lpToken = chef.lpToken(pid);
93      (uint128 enSushiPerShare, ) = chef.poolInfo(pid);
94      IERC20(lpToken).safeTransfer(msg.sender, amount);
95      uint stSushi = stSushiPerShare.mul(amount).divCeil(1e12);
96      uint enSushi = uint(enSushiPerShare).mul(amount).div(1e12);
97      if (enSushi > stSushi) {
98          sushi.safeTransfer(msg.sender, enSushi.sub(stSushi));
99      }
100
101      IRewarder rewarder = chef.rewarder(pid);
102      (uint128 enRewardPerShare, ) = rewarder.poolInfo(pid);
103      (IERC20[] memory rewardTokens, ) = rewarder.pendingTokens(pid, address(this), 0);
104      uint stReward = stRewardPerShare[pid].mul(amount).divCeil(1e12);
105      uint enReward = uint(enRewardPerShare).mul(amount).div(1e12);
106      if (enReward > stReward) {
107          rewardTokens[0].safeTransfer(msg.sender, enReward.sub(stReward));

```

```

108     }
109     return pid;
110 }

```

Listing 3.8: WMiniChefV2::burn()

The same issue is also applicable to another wrapper, i.e., WStakingDualRewards.

Recommendation Maintain the proper accounting information for each token ID, instead of the pool id.

Status The issue has been fixed by this commit: 28e8c87.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: “Transfers `_value` amount of tokens to address `_to`, and *MUST* fire the Transfer event. The function *SHOULD* throw if the message caller’s account balance does not have enough tokens to spend.”

```

64     function transfer(address _to, uint _value) returns (bool) {
65         //Default assumes totalSupply can't be over max (2^256 - 1).
66         if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67             balances[msg.sender] -= _value;
68             balances[_to] += _value;
69             Transfer(msg.sender, _to, _value);
70             return true;
71         } else { return false; }
72     }

```

```

74     function transferFrom(address _from, address _to, uint _value) returns (bool) {
75         if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
76             balances[_to] += _value;
77             balances[_from] -= _value;
78             allowed[_from][msg.sender] -= _value;
79             Transfer(_from, _to, _value);
80             return true;
81         } else { return false; }
82     }

```

Listing 3.9: ZRX.sol

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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `claimRewards()` routine in the `QsQuickLpDelegate` contract. If the USDT token is supported as token, the unsafe version of `EIP20Interface(token).transfer(account, accrued)` (line 145) may revert as there is no return value in the USDT token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```

120     function claimRewards(address account) public returns (uint) {
121         claimFromQuick();
122
123         updateLPSupplyIndex();
124         updateSupplierIndex(account);
125
126         mintToFilda();
127
128         // Get user's token accrued.
129         for (uint8 i = 0; i < rewardsTokens.length; i++) {
130             address token = rewardsTokens[i];
131
132             uint accrued = tokenUserAccrued[account][token];
133             if (accrued == 0) continue;
134
135             lpSupplyStates[token].balance = sub_(lpSupplyStates[token].balance, accrued)
136             ;
137
138             if (rewardsFToken[token] != address(0)) {
139                 uint err = CErc20(rewardsFToken[token]).redeemUnderlying(accrued);
140                 require(err == 0, "redeem fmdx failed");
141             }
142
143             // Clear user's token accrued.
144             tokenUserAccrued[account][token] = 0;

```

```

145         EIP20Interface(token).transfer(account, accrued);
146     }
147
148     return 0;
149 }

```

Listing 3.10: QsQuickLpDelegate::claimRewards()

This issue is present in other contracts, including QsMdxDelegate, QsSushiLPDelegate, QsQuickLPDelegate, QsQuickDualLPDelegate().

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom(). Note the safeApprove() counterpart may need to invoke twice: the first time resets the allowance to 0 and the second time sets the intended spending allowance.

Status The issue has been fixed by this commit: f848a69.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Fi1DA protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

130     function _setCreditLimit(address protocol, uint creditLimit) public {
131         require(msg.sender == owner(), "only owner can set protocol credit limit");
132
133         creditLimits[protocol] = creditLimit;
134         emit CreditLimitChanged(protocol, creditLimit);
135     }
136
137     function _setCompToken(address _compToken) public onlyOwner {
138         address oldCompToken = compToken;
139         compToken = _compToken;
140         emit NewCompToken(oldCompToken, compToken);
141     }
142

```

```

143     function _setSafetyVault(address _safetyVault) public onlyOwner {
144         address oldSafetyVault = safetyVault;
145         safetyVault = _safetyVault;
146         emit NewSafetyVault(oldSafetyVault, safetyVault);
147     }
148
149     function _setSafetyVaultRatio(uint _safetyVaultRatio) public onlyOwner {
150         uint oldSafetyVaultRatio = safetyVaultRatio;
151         safetyVaultRatio = _safetyVaultRatio;
152         emit NewSafetyVaultRatio(oldSafetyVaultRatio, safetyVaultRatio);
153     }

```

Listing 3.11: Example Setters in the QsConfig Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

```

7  contract TransparentUpgradeableProxyImpl is TransparentUpgradeableProxy {
8      constructor(
9          address _logic,
10         address _admin,
11         bytes memory _data
12     ) public payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
13 }

```

Listing 3.12: TransparentUpgradeableProxyImpl::constructor()

Recommendation Promptly transfer the privileged account 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 with the team. For the time being, the team has clarified the plan to migrate the `owner` account to a trusted multi-sig account with necessary timelock.

4 | Conclusion

In this audit, we have analyzed the `FilDA` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and leveraged farming. The protocol designs are architected and forked based on `Compound` and `Alpha HomoraV2` and synced into the `FilDA` platform to capitalize the benefits of both systems. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. <https://cwe.mitre.org/data/definitions/628.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.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: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.