# SMART CONTRACT AUDIT REPORT

for

# SparkleX Earning

Prepared By: Xiaomi Huang

PeckShield

July 10, 2025

## Document Properties

| | |
|---|---|
| Client | SparkleX |
| Title | Smart Contract Audit Report |
| Target | SparkleX Earning |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Matthew Jiang, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | July 10, 2025 | Xuxian Jiang | Final Release |
| 1.0-rc1 | July 5, 2025 | Xuxian Jiang | Release Candidate #1 |

## 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** 4

1.1 About SparkleX Earning . . . . 4

1.2 About PeckShield . . . . 5

1.3 Methodology . . . . 5

1.4 Disclaimer . . . . 7

**2 Findings** 9

2.1 Summary . . . . 9

2.2 Key Findings . . . . 10

**3 Detailed Results** 11

3.1 Revisited Borrow Amount Calculation For Leverage/Deleverage . . . . 11

3.2 Timely Management Fee Collection in SparkleXVault . . . . 13

3.3 Enhanced Token Allowance Management in BaseAAVEStrategy . . . . 14

3.4 Possibly Blocked EtherFi Withdrawal in EtherFiHelper . . . . 15

3.5 Revisited Slippage Control in ETHEtherFiAAVEStrategy::_swapUsingCurve() . . . . 16

3.6 Trust Issue of Admin Keys . . . . 17

**4 Conclusion** 20

**References** 21

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `SparkleX Earning` protocol, 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 is well implemented with extensive documentation. This document outlines our audit results.

## 1.1 About SparkleX Earning

`SparkleX Earning` protocol provides users with automated tools to optimize yield farming strategies and earn the best possible returns on their crypto assets with minimal effort. It follows the classic `ERC4626` vault design and has the built-in support of three yield strategies, i.e., `ETHEtherFiAAVEStrategy`, `PendleAAVEStrategy`, and `PendleStrategy`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SparkleX Earning

| Item | Description |
|---|---|
| Name | SparkleX |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | July 10, 2025 |

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

- https://github.com/sparklexai/earning.git (7d8c640)

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

- https://github.com/sparklexai/earning.git (90ac585)

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

## 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

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 |

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 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.

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. |

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `SparkleX Earning` 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 | 0 | |
| Medium | 3 | ■■■ |
| Low | 3 | ■■■ |
| Informational | 0 | |
| Total | 6 | |

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 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key SparkleX Earning Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Revisited Borrow Amount Calculation For Leverage/Deleverage | Business Logic | Resolved |
| PVE-002 | Medium | Timely Management Fee Collection in SparkleXVault | Business Logic | Resolved |
| PVE-003 | Low | Enhanced Token Allowance Management in BaseAAVEStrategy | Business Logic | Resolved |
| PVE-004 | Medium | Possibly Blocked EtherFi Withdrawal in EtherFiHelper | Business Logic | Resolved |
| PVE-005 | Low | Revisited Slippage Control in ETHEtherFiAAVEStrategy::_swapUsingCurve() | Time And State | Resolved |
| PVE-006 | Medium | Trust Issue of Admin Keys | Security Features | 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 Revisited Borrow Amount Calculation For Leverage/Deleverage

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `BaseAAVEStrategy`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

Among the three built-in strategies, two of them are related to the `Aave` protocol and a common `BaseAAVEStrategy` base contract is created to facilitate the strategy functionality management. In the process of examining the logic to compute the borrow amount for leverage/deleverage operations, we notice the amount calculation needs to be revisited.

To elaborate, we show below the code snippet of the related `_leveragePosition()` routine, which is used to complete the position leveraging in `Aave` based on the given asset token amount. Specifically, the new borrow amount is calculated via the internal helper routine `AAVEHelper(_aaveHelper).previewLeverageForInvest()`, which is given two parameters, i.e., `_capAmountByBalance(_asset, _assetAmount, false)` and `_borrowAmount` (line 196). Since the input `_assetAmount` has been converted to the supply token (line 194), the first parameter should be simply `0`, not `_capAmountByBalance(_asset, _assetAmount, false)` (line 196).

```
190     function _leveragePosition(uint256 _assetAmount, uint256 _borrowAmount, bytes memory
            _extraAction)
191         internal
192         virtual
193     {
194         _prepareSupplyFromAsset(_assetAmount, _extraAction);
195         uint256 _toBorrow = AAVEHelper(_aaveHelper).previewLeverageForInvest(
196             _capAmountByBalance(_asset, _assetAmount, false), _borrowAmount
197         );
```

```

        // use flashloan to leverage position
        (, address _flProvider,) = AAVEHelper(_aaveHelper).useSparkFlashloan();
        IPool(_flProvider).flashLoanSimple(
            address(this),
            address(AAVEHelper(_aaveHelper)._borrowToken()),
            _toBorrow,
            abi.encode(true, 0, _extraAction),
            0
        );
    }
```

Listing 3.1: `::_leveragePosition()`

Moreover, another related `previewCollect()` routine is used to estimate the amount required to adjust the position leverage. This routine can be improved by addressing the flashloan-assisted deleverage logic. In particular, the given `_amountToCollect` variable needs to be adjusted as `_amountToCollect - residue` (lines 318, 320, and 326).

```
310         // case [3] deleverage using flashloan
311         (uint256 _netSupplyAsset, uint256 _debtAsset,) = BaseAAVEStrategy(_strategy).
                getNetSupplyAndDebt(true);
312         uint256 _threshold = applyLeverageMargin(_netSupplyAsset);
313         _result = new uint256[](5);
314         _result[0] = 3;
315         _result[1] = _netSupplyAsset;
316         // borrow amount for full deleverage to repay entire debt
317         _result[2] = TokenSwapper(BaseAAVEStrategy(_strategy)._swapper()).
                applySlippageMargin(_debtAsset);
318         if (_amountToCollect < _threshold) {
319             // borrow amount for partial deleverage to repay a portion of debt
320             _result[3] = getMaxLeverage(_amountToCollect);
321             _result[3] = _result[3] > _debtAsset ? _result[2] : _result[3];
322             (,, uint256 _flFee) = useSparkFlashloan();
323             _result[4] = _result[3] == _result[2]
324                 ? _totalInSupply
325                 : BaseAAVEStrategy(_strategy)._convertBorrowToSupply(
326                     _result[3] + (_result[3] * _flFee / Constants.TOTAL_BPS) +
                        _amountToCollect
327                 );
328         } else {
329             _result[3] = _result[2];
330             _result[4] = _totalInSupply;
331         }
```

Listing 3.2: `AAVEHelper::previewCollect()`

**Recommendation** Revisit the above-mentioned routines to properly compute the borrow amount for leverage/deleverage.

**Status** The issue has been fixed by the following commits: `d9adb3a`.

## 3.2 Timely Management Fee Collection in SparkleXVault

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SparkleXVault`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `SparkleX Earning` protocol is designed to charge a management fee that is based on the total assets managed by the protocol. In the process of reviewing the management fee collection, we notice the logic should be improved.

In particular, in current implementation, the management fee is charged when the helper routine `_accumulateManagementFeeInternal()` is invoked in three different scenarios: (1) The first one is the vault is initialized; (2) The second one is the management fee percentage is updated; and (3) the redemption claimer or the owner explicitly calls to accumulate the management fee. To timely and fairly collect the management fee, this helper routine should be invoked upon the entry to deposit and withdrawal operation.

```
    function _withdraw(address caller, address receiver, address owner, uint256 assets,
        uint256 shares)
        internal
        override
        whenNotPaused
    {
        if (caller != owner) {
            _spendAllowance(owner, caller, shares);
        }
        _burn(owner, shares);
        assets = _chargeWithdrawFee(assets, owner);
        SafeERC20.safeTransfer(ERC20(asset()), receiver, assets);

        emit Withdraw(caller, receiver, owner, assets, shares);
    }
```

Listing 3.3: `SparkleXVault::_withdraw()`

**Recommendation** Revise the above-mentioned routines to properly charge the management fee.

**Status** The issue has been fixed by the following commits: `59a839e`.

## 3.3 Enhanced Token Allowance Management in BaseAAVEStrategy

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseAAVEStrategy`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The supported strategies may require the token transfers (or swaps) from one contract (token type) to another to create a leveraged position. With that, there is a need to properly manage the token allowance for swaps and transfers. Our analysis shows the token allowance can be improved.

To elaborate, we show below the implementation of the related `_prepareAllowanceForHelper()` function. As the name indicates, this function is used to approve the `AAVEHelper` contract to spend a number of tokens, including `supplyToken`, `borrowToken`, and `supplyAToken`. Our analysis shows the last two approvals (lines 59-60) are not necessary. In the meantime, where the `AAVEHelper` contract is updated, there is a need to revoke the allowance from the old `AAVEHelper` contract.

```
55    function _prepareAllowanceForHelper() internal {
56        _delegateCreditToHelper();
57        _approveToken(address(AAVEHelper(_aaveHelper)._supplyToken()), _aaveHelper);
58        _approveToken(address(AAVEHelper(_aaveHelper)._borrowToken()), _aaveHelper);
59        _approveToken(address(AAVEHelper(_aaveHelper)._supplyAToken()), _aaveHelper);
60        _approveToken(address(AAVEHelper(_aaveHelper)._borrowToken()), address(aavePool)
              );
61    }
```

Listing 3.4: `BaseAAVEStrategy::_prepareAllowanceForHelper()`

Further, the `AAVEHelper` contract has a related `_setTokensAndApprovals()` routine that can be similarly improved by removing the `supplyAtoken` approval. The `TokenSwapper` contract has two related functions, i.e., `_callPendleRouter()`, `swapExactInWithUniswap()`, and `swapInCurveTwoTokenPool()`, can be improved by resetting the allownace to zero after the swap.

**Recommendation** Revise the above-mentioned routines to properly manage the intended token allowance.

**Status** The issue has been fixed by the following commits: `e83f587`.

## 3.4 Possibly Blocked EtherFi Withdrawal in EtherFiHelper

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `EtherFiHelper`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `SparkleX Earning` protocol has a core helper contract, i.e., `EtherFiHelper`, to facilitate the interaction with the `EtherFi` protocol. Specifically, it handles the user deposit and withdraw requests. While reviewing the user withdrawal logic, we notices its implementation has an issue that may be abused to block legitimate withdrawals.

To elaborate, we show below the implementation of the related `requestWithdrawFromEtherFi()` routine. As the name indicates, it is used to make a request to withdraw user funds. We notice the active requests for withdrawals are capped at `MAX_ACTIVE_WITHDRAW`. As a result, a malicious user may repeatedly request to withdraw with a tiny amount so that the withdrawal queue is filled to block legitimate user withdrawals (line 96).

```
95      function requestWithdrawFromEtherFi(uint256 _toWithdrawWeETH, uint256 _swapLoss)
            external returns (uint256) {
96          if (activeWithdrawRequests >= MAX_ACTIVE_WITHDRAW) {
97              revert Constants.TOO_MANY_WITHDRAW_FOR_ETHERFI();
98          }
99
100         SafeERC20.safeTransferFrom(ERC20(address(weETH)), msg.sender, address(this),
                _toWithdrawWeETH);
101         uint256 _toWithdraw = weETH.unwrap(_toWithdrawWeETH);
102
103         uint256 _reqID = etherfiLP.requestWithdraw(address(this), _toWithdraw);
104         emit WithdrawRequestFromEtherFi(msg.sender, _reqID, _toWithdraw);
105
106         IWithdrawRequestNFT.WithdrawRequest memory _request = etherfiWithdrawNFT.
                getRequest(_reqID);
107         require(_request.isValid, "withdraw invalid!");
108         require(etherfiWithdrawNFT.ownerOf(_reqID) == address(this), "withdraw NFT owner
                !");
109
110         _updateWithdrawReqAccounting(msg.sender, _reqID, _swapLoss, false);
111         return _reqID;
112     }
```

Listing 3.5: `EtherFiHelper::requestWithdrawFromEtherFi()`

**Recommendation** Validate the user requests for withdrawal so that only legitimate users can be granted.

**Status** The issue has been fixed by the following commits: `00fec9c`.

## 3.5 Revisited Slippage Control in ETHEtherFiAAVEStrategy::_swapUsingCurve()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ETHEtherFiAAVEStrategy`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

### Description

The built-in strategies in `SparkleX Earning` has the constant need of swapping one asset to another. With that, the protocol has provided two helper routines to facilitate the asset conversion: `_swapUsingCurve()` and `_swapUsingUniswap()`

```
    function _swapUsingCurve(ERC20 _supplyToken, ERC20 _borrowToken, uint256 _expectOutAmount)
        internal
        returns (uint256, uint256)
    {
        uint256 _expectedIn = TokenSwapper(_swapper).queryXWithYInCurve(
            address(_supplyToken), address(_borrowToken), weETHPool, _expectOutAmount
        );
        uint256 _cappedIn = _capAmountByBalance(_supplyToken, _expectedIn, true);
        uint256 _actualOut = TokenSwapper(_swapper).swapInCurveTwoTokenPool(
            address(_supplyToken), address(_borrowToken), weETHPool, _cappedIn, _expectOutAmount
        );
        return (_cappedIn, _actualOut);
    }

    function _swapUsingUniswap(ERC20 _supplyToken, ERC20 _borrowToken, uint256 _expectOutAmount)
        internal
        returns (uint256, uint256)
    {
        (int256 _weETHToETHPrice, , uint8 _priceDecimal) = TokenSwapper(_swapper).getPriceFromChainLink(weETH_ETH_FEED);
        uint256 _expectedIn =
            _expectOutAmount * Constants.convertDecimalToUnit(_priceDecimal) / uint256(_weETHToETHPrice);
        uint256 _cappedIn = _capAmountByBalance(_supplyToken, _expectedIn, true);
```

```
        uint256 _actualOut = TokenSwapper(_swapper).swapExactInWithUniswap(
            address(_supplyToken), address(_borrowToken), weETHUniPool, _cappedIn,
                _expectOutAmount
        );
        return (_cappedIn, _actualOut);
    }
```

Listing 3.6: ETHEtherFiAAVEStrategy::_swapUsingCurve()

To elaborate, we show above these two helper routines. We notice the first routine in essence queries the `curveRouter` contract for the expected input token amount in order to receive the given output token amount. And the swap operation makes use of the calculated input token amount as the slippage control to avoid being sandwiched by a MEV bot. However, this is the proper approach to mitigate the MEV risk as it is still vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing a trusted oracle (or a `TWAP (time-weighted average price)` of a deep liquidity pool). Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

**Recommendation** Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of protocol users. Note this issue also affects other routines, including `_swapPTForRollOver()`, `_swapAssetForPT()`, and `_swapPTForAsset()` in the `PendleHelper` contract.

**Status** The issue has been fixed by the following commits: `28e9658`.

## 3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

### Description

In the `SparkleX Earning` protocol, there is a special administrative `owner` account. This administrative account plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, whitelist strategies, and collect fees). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function updateStrategyAllocation(address _strategy, uint256 _newAlloc) external onlyOwner {
        if (strategyAllocations[_strategy] == 0  _newAlloc == 0) {
            revert Constants.WRONG_STRATEGY_ALLOC_UPDATE();
        }
        strategyAllocations[_strategy] = _newAlloc;
        emit StrategyAllocationChanged(msg.sender, _strategy, _newAlloc);
    }

    function setRedemptionClaimer(address _newClaimer) external onlyOwner {
        if (_newClaimer == Constants.ZRO_ADDR) {
            revert Constants.INVALID_ADDRESS_TO_SET();
        }
        emit RedemptionClaimerChanged(_redemptionClaimer, _newClaimer);
        _redemptionClaimer = _newClaimer;
    }

    function setFeeRecipient(address _newRecipient) external onlyOwner {
        if (_newRecipient == Constants.ZRO_ADDR) {
            revert Constants.INVALID_ADDRESS_TO_SET();
        }
        emit FeeRecipientChanged(_feeRecipient, _newRecipient);
        _feeRecipient = _newRecipient;
    }
    function setEarnRatio(uint256 _ratio) external onlyOwner {
        if (_ratio > Constants.TOTAL_BPS) {
            revert Constants.INVALID_BPS_TO_SET();
        }
        EARN_RATIO_BPS = _ratio;
        emit EarnRatioChanged(msg.sender, _ratio);
    }

    function setWithdrawFeeRatio(uint256 _ratio) external onlyOwner {
        if (_ratio >= Constants.TOTAL_BPS) {
            revert Constants.INVALID_BPS_TO_SET();
        }
        WITHDRAW_FEE_BPS = _ratio;
        emit WithdrawFeeChanged(msg.sender, _ratio);
    }

    function setManagementFeeRatio(uint256 _ratio) external onlyOwner {
        if (_ratio >= Constants.TOTAL_BPS) {
            revert Constants.INVALID_BPS_TO_SET();
        }
        _accumulateManagementFeeInternal();
        MANAGEMENT_FEE_BPS = _ratio;
        emit ManagementFeeChanged(msg.sender, _ratio);
    }
```

Listing 3.7: Example Privileged Operations in `Manager`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

**Recommendation** Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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** The issue has been mitigated with the use of a multi-sig account to hold the admin account.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the `SparkleX Earning` protocol, which provides users with automated tools to optimize yield farming strategies and earn the best possible returns on their crypto assets with minimal effort. It follows the classic `ERC4626` vault design and has the built-in support of three yield strategies, i.e., `ETHEtherFiAAVEStrategy`, `PendleAAVEStrategy`, and `PendleStrategy`. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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