SMART CONTRACT AUDIT REPORT

for

SparkleX UniswapV3 LP Strategy

Prepared By: Xiaomi Huang

PeckShield

October 3, 2025

Document Properties

Client	SparkleX
Title	Smart Contract Audit Report
Target	SparkleX UniswapV3 LP Strategy
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	October 3, 2025	Xuxian Jiang	Final Release
1.0-rc1	October 3, 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

Given the opportunity to review the design document and related smart contract source code of the new `UniswapV3 LP` strategy in `SparkleX`, 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 a number of yield strategies. This audit focuses on the new `UniswapV3`-related strategy. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SparkleX UniswapV3 LP Strategy

Item	Description
Name	SparkleX
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 3, 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 (3bd004e)

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 (2b2aa1b)

1.2 About PeckShield

PeckShield Inc. [5] 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 [4]:

- 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) [3], 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 `UniswapV3`-related strategy in `SparkleX`. 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	1	■
Low	2	■ ■
Informational	0	
Total	3	

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 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key SparkleX UniswapV3 LP Strategy Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Asset Type Confusion in BaseAAVEStrategy	Business Logic	Acknowledged
PVE-002	Medium	Inaccurate Position Amount Calculation in UniV3LPFarmingStrategy	Business Logic	Resolved
PVE-003	Low	Improved TWAP Price Calculation in LPFarmingHelper	Coding Practices	Resolved

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 Possible Asset Type Confusion in BaseAAVEStrategy

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseAAVEStrategy`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The supported strategies have a common base contract, i.e., `BaseAAVEStrategy`, which provides a number of helper functions. One of them is `_deleverageByFlashloan()` to deleverage the position via flashloans. In the process of reviewing the deleverage logic, we notice an issue that may confuse the token types being used.

To elaborate, we show below the implementation of this `_deleverageByFlashloan()` function. It takes five arguments: `_netSupplyAsset`, `_debtAsset`, `_expectedAsset`, `_deleveragedAmount`, and `_extraAction`. The first four arguments are token amounts denominated in the strategy asset. However, when the second argument `_debtAsset` and the fourth argument `_deleveragedAmount` are used as flashloan arguments (lines 239 and 248), they are directly used as the borrow token. The conversion from strategy asset to the borrow token requires the use of `_convertAssetToBorrow()`.

```
226     function _deleverageByFlashloan(
227         uint256 _netSupplyAsset,
228         uint256 _debtAsset,
229         uint256 _expectedAsset,
230         uint256 _deleveragedAmount,
231         bytes calldata _extraAction
232     ) internal {
233         (, address _flProvider,) = AAVEHelper(_aaveHelper).useSparkFlashloan();
234         if (_expectedAsset > 0 && _expectedAsset < AAVEHelper(_aaveHelper).
                applyLeverageMargin(_netSupplyAsset)) {
235             // deleverage a portion if possible
236             IPool(_flProvider).flashLoanSimple(
```

```
                address(this),
                address(AAVEHelper(_aaveHelper)._borrowToken()),
                _deleveragedAmount,
                abi.encode(false, _expectedAsset, _extraAction),
                0
            );
        } else {
            // deleverage everything
            IPool(_flProvider).flashLoanSimple(
                address(this),
                address(AAVEHelper(_aaveHelper)._borrowToken()),
                _debtAsset,
                abi.encode(false, 0, _extraAction),
                0
            );
        }
    }
```

Listing 3.1: `BaseAAVEStrategy::_deleverageByFlashloan()`

Recommendation Revise the above `_deleverageByFlashloan()` routine to properly convert the asset amount to the borrow amount. Also, there is another function `AAVEHelper:previewCollect()` that shares a similar issue.

Status The issue has been acknowledged.

3.2 Inaccurate Position Amount Calculation in UniV3LPFarmingStrategy

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `UniV3LPFarmingStrategy`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The `SparkleX Earning` protocol provides a new `UniswapV3`-based strategy, i.e., `UniV3LPFarmingStrategy`. By providing in-range liquidity, the new strategy allows to earn yields from the swap fee and possibly compound the liquidity position value. While reviewing the logic to calculate the position value, we notice current implementation should be improved.

In particular, we show below the implementation of the `_getPositionAmountWithCurrentX96()` routine. As the name indicates, this routine is designed to compute the position amount in terms of the pool's `token0` and `token1`. For each composite token in the pool, we need to compute the principal amount as well as the accumulated fee. It comes to our attention that the accumulated fee only includes the last computed fee (line 420), missing the new fee that may be accumulated ever since.

```
410     function getPositionAmountWithCurrentX96(uint160 _currentSqrtPX96, LPPositionInfo
            calldata _positionInfo)
411         public
412         view
413         returns (uint256, uint256)
414     {
415         if (!_positionInfo.active) {
416             return (0, 0);
417         } else {
418             address _posMgr = getDexPositionManager();
419             (uint256 _token0Fee, uint256 _token1Fee) =
420                 UniV3PositionMath.getLastComputedFees(_posMgr, _positionInfo.tokenId);
421             (uint256 _token0InLp, uint256 _token1InLp) =
422                 UniV3PositionMath.getAmountsForPosition(_posMgr, _positionInfo.tokenId,
                        _currentSqrtPX96);
423             return (_token0Fee + _token0InLp, _token1Fee + _token1InLp);
424         }
425     }
```

Listing 3.2: `UniV3LPFarmingStrategy::getPositionAmountWithCurrentX96()`

Recommendation Revise the above-mentioned routine to ensure the accumulated fee is current.

Status The issue has been fixed by the following commits: `2b2aa1b`.

3.3 Improved TWAP Price Calculation in LPFarmingHelper

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPFarmingHelper`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

To facilitate the interaction with external `UniswapV3` pools, `SparkleX Earning` has a helper contract `LPFarmingHelper`, which includes the logic to compute the pool `TWAP`-based oracle price in the `sqrtX96` -compatible manner. Our analysis shows a minor improvement to current implementation.

In particular, we show below the implementation of a related routine, i.e., `_getTwapPriceInSqrtX96 ()`. The time-weighted average price is computed as `(tickCumulatives[1] - tickCumulatives[0])/ twapInterval` (lines 270-273), which may be optimized to round to negative infinity if `tickCumulatives [1] < tickCumulatives[0]`. In fact, an example use can be found in the `periphery/libraries/OracleLibrary` contract from the `UniswapV3-periphery` repository.

```
    function getTwapPriceInSqrtX96(address _pool, uint32 _twapInterval) public view returns (uint160, int24) {
        if (_twapInterval == 0) {
            revert Constants.WRONG_TWAP_OBSERVE_INTERVAL();
        }

        uint32[] memory secondsAgos = new uint32[](2);
        secondsAgos[0] = _twapInterval; // from (before)
        secondsAgos[1] = 0; // to (now)

        (int56[] memory tickCumulatives,) = IUniswapV3PoolDerivedState(_pool).observe(secondsAgos);
        int56 _tickCumulativeDiff = tickCumulatives[1] - tickCumulatives[0];
        int56 _timeDiff = int56(int32(_twapInterval));
        int24 _tickTwap = int24(_tickCumulativeDiff / _timeDiff);
        return (TickMath.getSqrtRatioAtTick(_tickTwap), _tickTwap);
    }
```

Listing 3.3: `LPFarmingHelper::getTwapPriceInSqrtX96()`

Recommendation Improve the above `getTwapPriceInSqrtX96()` function to better make use of the time-weighted average price.

Status The issue has been fixed by the following commits: `2b2aa1b`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `UniswapV3`-related strategy in `SparkleX`, which allows 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 a number of yield strategies. This audit focuses on the new `UniswapV3` strategy. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[2] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[3] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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