SMART CONTRACT AUDIT REPORT

for

SparkleX Farming

Prepared By: Xiaomi Huang

PeckShield

June 26, 2025

Document Properties

Client	SparkleX
Title	Smart Contract Audit Report
Target	SparkleX Farming
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	June 26, 2025	Xuxian Jiang	Final Release
1.0-rc1	June 23, 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 SparkleX Farming 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 Farming

SparkleX Farming protocol provides each user with a secure and efficient vault to handle all operations related to Uniswap V3 LP. Additionally, we leverage an AI agent to help users optimize and manage their positions, improving capital efficiency and maximizing returns. To protect user funds, the contract enforces strict limits on what the agent is allowed to do. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SparkleX Farming

Item	Description
Name	SparkleX
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 26, 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/farming.git (ba68c2b)

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

- https://github.com/sparklexai/farming.git (9bcc893)

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

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

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 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 Farming` 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	1	🟦
Informational	1	🟦
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 3 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key SparkleX Farming Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Swap Amount Calculation in LiqMath	Business Logic	Resolved
PVE-002	Informational	Lack of Agent Authorization/Strategy Validation in UserVault	Security Features	Resolved
PVE-003	Medium	Necessity of User Refund in UniswapV3Mint	Business Logic	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	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 Swap Amount Calculation in LiqMath

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

Description

The protocol has a core `LiqMath` library contract that is designed to compute the optimal swap amount used in adding single-side liquidity or providing imbalanced token amounts. In the process of examining the logic to compute the token swap amount, we notice the out-of-range calculation needs to be revisited.

To elaborate, we show below the code snippet of the related `getToken0SwapAmount()` routine. Given total amount of `token0`, It is used to calculate the amount to swap. Our analysis shows that when current price is below or equal to lower bound, we only need to provide `token0`. In other words, the swap amount should be `0`, not current `totalAmount` (line 34). Similarly, when current price is above or equal to upper bound, we only need to provide `token1`. In other words, the swap amount should be `totalAmount`, not current `0` (line 39). Note the `getToken1SwapAmount()` routine should be similarly fixed.

```
25      function getToken0SwapAmount(
26          uint160 sqrtPriceX96,
27          uint160 sqrtPriceLowerX96,
28          uint160 sqrtPriceUpperX96,
29          uint256 totalAmount
30      ) internal pure returns (uint256 swapAmount) {
31          require(sqrtPriceLowerX96 < sqrtPriceUpperX96, "Invalid price range");
32          // If current price is below or equal to lower bound
33          if (sqrtPriceX96 <= sqrtPriceLowerX96) {
34              swapAmount = totalAmount;
35              return swapAmount;
```

```
        }
        // If current price is above or equal to upper bound
        if (sqrtPriceX96 >= sqrtPriceUpperX96) {
            swapAmount = 0;
            return swapAmount;
        }

        uint256 ratio = _ratio(
            sqrtPriceX96,
            sqrtPriceLowerX96,
            sqrtPriceUpperX96
        );
        uint256 priceX96 = FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, 1 << 96);
        swapAmount = (totalAmount * ratio) / (priceX96 + ratio);
    }
```

Listing 3.1: `LiqMath::getToken0SwapAmount()`

Moreover, we notice the swap amount calculation ignores possible price impact from the swap. An improved approach is to follow the algorithm outlined here, i.e., https://www.desmos.com/calculator/oiv0rti0ss. It is used to swap-then-deposit the maximum liquidity to a particular range on the same `UniswapV3` pool, given some initial starting tokens.

Recommendation Revisit the above-mentioned routines to properly compute the required swap amount before adding the intended liquidity.

Status The issue has been fixed by the following commits: `da99440`, `8837727`, `79e7308`, `fd978b0`, and `58c59c4`.

3.2 Lack of Agent Authorization/Strategy Validation in UserVault

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UserVault`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The protocol creates a user-specific vault to hold liquidity positions. By design, the vault allows to add a user agent to operate on behalf of the user with a number of approved strategies. While reviewing the main interaction logic with supported strategies, we notice current logic does not authorize the agent as intended and can also be improved by validating the provided strategy.

To elaborate, we show below the implementation of the related `work()` function. It has a rather straightforward logic in recording the calling user, setting up the position and strategy, and then invoking the strategy execution. However, the user agent can not directly call it as the modifier `onlyManager` allows the direct caller to be user and manager only. Also, the given strategy can be validated with the manager to ensure it is approved.

```
    function work(
        address _caller,
        uint256 _positionID,
        address _strategy,
        bytes calldata _data
    ) external onlyManager {
        Position storage _pos;
        if (_positionID == 0) {
            _positionID = nextPositionId;
            nextPositionId += 1;
            _pos = _positions[_positionID];
        } else {
            _pos = _positions[_positionID];
            if (_positionID >= nextPositionId) revert BadPositionID();
        }

        // Set the execution scope
        STRATEGY = _strategy;
        POSITION_ID = _positionID;

        // If the agent or user calls this function directly,
        // we need to set the caller to the agent or user manually.
        if (msg.sender != manager) {
            _caller = msg.sender;
        }

        // Execute the strategy
        (uint8 posType, bytes memory posData) = IStrategy(_strategy).execute(
            _caller,
            _positionID,
            _data
        );
        _pos.posType = posType;
        _pos.data = posData;

        // Reset the execution scope
        POSITION_ID = _NO_ID;
        STRATEGY = _NO_ADDRESS;

        emit DoWork(_caller, _strategy);
    }
```

Listing 3.2: `UserVault::work()`

Recommendation Revise the above routine to ensure the agent is allowed and the given strategy is already approved.

Status The issue has been resolved as it is part of the design in allowing only the user to directly call the above `work()` function, as explicitly commented here: `f252cb2`.

3.3 Necessity of User Refund in UniswapV3Mint

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `UniswapV3Mint`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

As mentioned earlier, the protocol provides a number of strategies so that user vaults can interact directly. In the process of analyzing a specific strategy `UniswapV3Mint`, we notice current logic does not refund users with the remaining balances, if any.

To elaborate, we show below the implementation of the related `execute()` function. It has a rather straightforward logic in validating the input arguments, requesting user funds, and then creating a new liquidity position. However, it comes to our attention that it does not return back remaining user funds, if any.

```
    function execute(
        address _caller,
        uint256 _positionID,
        bytes calldata _data
    )
        external
        override
        nonReentrant
        returns (uint8 posType, bytes memory posData)
    {
        Position memory _position = IUserVault(msg.sender).positions(_positionID);
        if (_position.data.length != 0) revert PositionAlreadyExists();

        (
            bool _userFund,
            INonfungiblePositionManager.MintParams memory _params
        ) = abi.decode(
                _data,
                (bool, INonfungiblePositionManager.MintParams)
            );
        if (_params.token0 >= _params.token1) revert InvalidTokenOrder();

        if (!_validateAgent(_caller, _userFund)) {
            revert NotAuthorized();
        }

        SafeERC20.safeIncreaseAllowance(
            IERC20(_params.token0),
            positionManager,
            _params.amount0Desired
        );
        SafeERC20.safeIncreaseAllowance(
            IERC20(_params.token1),
            positionManager,
            _params.amount1Desired
        );

        if (_userFund) {
            IUserVault(msg.sender).requestFundsFromUser(
                _params.token0,
                _params.amount0Desired
            );
            IUserVault(msg.sender).requestFundsFromUser(
                _params.token1,
                _params.amount1Desired
            );
        } else {
            IUserVault(msg.sender).requestFunds(
                _params.token0,
                _params.amount0Desired
            );
            IUserVault(msg.sender).requestFunds(
                _params.token1,
                _params.amount1Desired
            );
        }

        _params.recipient = msg.sender;
        _params.deadline = block.timestamp;

        (uint256 _tokenID, uint128 _liquidity, uint256 _amount0, uint256 _amount1) =
            INonfungiblePositionManager(positionManager)
            .mint(_params);

        emit Mint(
            msg.sender,
            _positionID,
            _tokenID,
            _params.token0,
            _params.token1,
            _liquidity,
            _amount0,
            _amount1
        );
        return (
            uint8(PositionType.V3_LP),
            abi.encode(
                V3Position({
                    tokenId: _tokenID,
                    token0: _params.token0,
                    token1: _params.token1,
                    fee: _params.fee
                })
            )
        );
    }
```

Listing 3.3: `UniswapV3Mint::execute()`

Recommendation Revise the above routine to return back any remaining funds to the user.

Status The issue has been fixed by this commit: `de66f71`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasisStrategy`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [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 `approve()` routine and analyze 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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 `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In current implementation, if we examine the `UniswapV3AddLiquidity::execute()` function that is designed to add liquidity into an existing position, in order to accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (lines 104-105).

```
93          (uint256 _t0Amount, uint256 _t1Amount) = _swapIfNeeded(
94              _v3Position.token0,
95              _v3Position.token1,
            _params.amount0Desired,
            _params.amount1Desired,
            _v3Position.fee,
            _v3Position.tokenId,
            _params.token0SwapPath,
            _params.token1SwapPath
        );

        IERC20(_v3Position.token0).approve(positionManager, _t0Amount);
        IERC20(_v3Position.token1).approve(positionManager, _t1Amount);

        INonfungiblePositionManager.IncreaseLiquidityParams
            memory _decParams = INonfungiblePositionManager
                .IncreaseLiquidityParams({
                    tokenId: _v3Position.tokenId,
                    amount0Desired: _t0Amount,
                    amount1Desired: _t1Amount,
                    amount0Min: _params.amount0Min,
                    amount1Min: _params.amount1Min,
                    deadline: block.timestamp
                });
        (uint128 _liquidity, uint256 _amount0, uint256 _amount1) =
            INonfungiblePositionManager(positionManager)
            .increaseLiquidity(_decParams);
```

Listing 3.6: `UniswapV3AddLiquidity::execute()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status The issue has been fixed by this commit: `e5a76ec`.

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 [2]

Description

In the `OTC Marketplace` 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., whitelist strategies, and upgrade contracts). 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.

```
    /// @notice Admin function to set pool whitelist for token pairs
    function setApprovedPools(bytes32[] calldata pools, bool allowed) external onlyOwner
         {
        for (uint256 i = 0; i < pools.length; i++) {
            approvedPools[pools[i]] = allowed;
            emit UpdateApprovedPool(pools[i], allowed);
        }
    }
    ...

    /// @notice Admin function to set strategy whitelist
    function setApprovedStrategies(address[] calldata strategies, bool allowed) external
         onlyOwner {
        for (uint256 i = 0; i < strategies.length; i++) {
            approvedStrategies[strategies[i]] = allowed;
            emit UpdateApprovedStrategy(strategies[i], allowed);
        }
    }
```

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.

Moreover, it should be noted that current contracts are to be deployed behind a proxy with a backend implementation. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

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

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SparkleX Farming` protocol, which provides each user with a secure and efficient vault to handle all operations related to `Uniswap V3 LP`. Additionally, we leverage an `AI` agent to help users optimize and manage their positions, improving capital efficiency and maximizing returns. To protect user funds, the contract enforces strict limits on what the agent is allowed to do. 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-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-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.