SMART CONTRACT AUDIT REPORT

for

Hegic Herge Protocol Upgrade

Prepared By: Xiaomi Huang

PeckShield

October 18, 2022

Document Properties

Client	Hegic
Title	Smart Contract Audit Report
Target	Hegic Herge Protocol Upgrade
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 18, 2022	Stephen Bie	Final Release
1.0-rc	October 14, 2022	Stephen Bie	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Hegic Herge` 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 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 Hegic

The `Hegic` protocol is an on-chain peer-to-pool options trading protocol built on Ethereum. With the protocol, DeFi and crypto users can trade 24/7, cash-settled, various on-chain ETH and WBTC option trading strategies with no KYC or registration required for trading. It provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current DeFi ecosystem.

Table 1.1: Basic Information of Hegic Herge Protocol Upgrade

Item	Description
Target	Hegic Herge Protocol Upgrade
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 18, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note this audit only covers the `contracts/packages/herge` sub-directory.

- https://github.com/hegic/contracts.git (3a0b690)

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

- https://github.com/hegic/contracts.git (a46c922)

1.2 About PeckShield

PeckShield Inc. [12] 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 OWASP Risk Rating Methodology [11]:

- 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 list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
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

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) [10], 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.

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 Logics	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 Hegic Herge protocol implementation. 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	1	■
Informational	2	■ ■
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 that 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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Hegic Herge Protocol Upgrade Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly *share* from Improper Deposit Initialization	Time and State	Fixed
PVE-002	High	Revisited Logic of CoverPool::_startNextEpoch()	Business Logic	Fixed
PVE-003	High	Revisited Logic of HegicStrategy::_create()	Business Logic	Fixed
PVE-004	Informational	Immutable States If Only Set at Constructor()	Coding Practices	Fixed
PVE-005	Informational	Suggested Event Generation for Key Operations	Coding Practices	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Costly *share* from Improper Deposit Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CoverPool`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [2]

Description

The `CoverPool` contract allows users to deposit the supported `coverToken` token and get in return `shares` to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the related code snippet of the `CoverPool` contract. The `provide()` routine is used for participating users to deposit the supported asset. In particular, inside the `provide()` routine, the internal `_provide()` routine is called to calculate the `share` amount of the deposit. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function provide(uint256 amount, uint256 positionId)
        external
        override
        returns (uint256)
    {
        if (positionId == 0) {
            positionId = _nextPositionId++;
            _mint(msg.sender, positionId);
        }
        require(
            _isApprovedOrOwner(msg.sender, positionId),
            "Yuo are has no access to this position"
        );
        require(
            windowSize > block.timestamp - epoch[currentEpoch].start,
```

```
103                "Enterence window is closed"
104            );
105            _bufferUnclaimedProfit(positionId);
106            uint256 shareOfProvide = _provide(positionId, amount);
107            coverToken.safeTransferFrom(msg.sender, address(this), amount);
108            // TODO emit Provided(positionId, amount, shareOfProvide, shareOf[positionId],
                   totalShare);
109            return positionId;
110        }
111
112        function _provide(uint256 positionId, uint256 amount)
113            internal
114            returns (uint256 shareOfProvide)
115        {
116            uint256 totalCoverBalance = coverTokenTotal();
117            shareOfProvide = totalCoverBalance > 0
118                ? (amount * totalShare) / totalCoverBalance
119                : amount;
120            shareOf[positionId] += shareOfProvide;
121            totalShare += shareOfProvide;
122        }
```

Listing 3.1: `CoverPool::provide()`

Specifically, when the pool is being initialized, the `shareOfProvide` directly takes the value of `amount` (line 119), which is under control by the malicious actor. As this is the first deposit, the current total share equals the calculated `shareOfProvide = totalCoverBalance > 0 ? (amount * totalShare)/ totalCoverBalance : amount = 1WEI`. With that, the actor can further transfer a huge amount of `coverToken` token to `CoverPool` contract with the goal of making the `pool share` extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool shares for the deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool shares.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `_provide()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first stake to avoid being manipulated.

Status The issue has been addressed by the following commit: `7e9cea0`.

3.2 Revisited Logic of CoverPool::_startNextEpoch()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: CoverPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the Hegic Herge protocol, the CoverPool contract implements an incentive mechanism that rewards the staking of the supported coverToken token with the profitToken token. In particular, the internal _startNextEpoch() routine called inside the privilege fixProfit() routine is used to settle the last reward epoch and start a new reward epoch. While examining its logic, we observe its current implementation should be improved.

To elaborate, we show below the related code snippet of the CoverPool contract. By design, the epoch[currentEpoch].cumulativePoint records the accumulated rewards per share at the beginning of the currentEpoch epoch and the cumulativeProfit records the latest accumulated rewards per share (i.e., the accumulated rewards per share at the end of the currentEpoch epoch). Inside the _startNextEpoch() routine, the statement of `uint256 profitOut = totalShare == 0 ? 0 : ((cumulativeProfit - epoch[currentEpoch].cumulativePoint)* coverTokenTotal())/ ADDITIONAL_DECIMALS` (line 307) is designed to calculate the total rewards shared by the total withdrawal in the currentEpoch epoch. Apparently, the current implementation does not meet the requirement. Given this, we suggest to improve the implementation as below: `uint256 profitOut = totalShareOut == 0 ? 0 : ((cumulativeProfit - epoch[currentEpoch].cumulativePoint)* totalShareOut)/ ADDITIONAL_DECIMALS` (line 307).

Moreover, inside the _startNextEpoch() routine, it comes to our attention that the totalShare is updated (line 314) but the corresponding coverToken balance (i.e., coverTokenTotal()) is not, which will make the pool share expensive and require necessary revision.

```
287     function fixProfit() external onlyRole(DEFAULT_ADMIN_ROLE) {
288         uint256 profitAmount = profitToken.balanceOf(address(this)) -
289             profitTokenBalance;
290         profitTokenBalance += profitAmount;
291         cumulativeProfit += (profitAmount * ADDITIONAL_DECIMALS) / totalShare;
292 
293         _startNextEpoch();
294         emit Profit(currentEpoch, profitAmount);
295     }
296 
297     function _startNextEpoch() internal {
298         require(
```

```
            MINIMAL_EPOCH_DURATION <
                block.timestamp - epoch[currentEpoch].start,
            "The epoch is too short to be closed"
        );
        uint256 totalShareOut = epoch[currentEpoch].totalShareOut;
        uint256 coverTokenOut = totalShare == 0
            ? 0
            : (totalShareOut * coverTokenTotal()) / totalShare;
        uint256 profitOut = totalShare == 0
            ? 0
            : ((cumulativeProfit - epoch[currentEpoch].cumulativePoint) *
                coverTokenTotal()) / ADDITIONAL_DECIMALS;

        epoch[currentEpoch].coverTokenOut = coverTokenOut;
        epoch[currentEpoch].profitTokenOut = profitOut;
        totalShare -= epoch[currentEpoch].totalShareOut;

        ...
    }
```

Listing 3.2: `CoverPool::fixProfit()&&_startNextEpoch()`

Recommendation Correct the implementation of the `_startNextEpoch()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `7e9cea0`.

3.3 Revisited Logic of HegicStrategy::_create()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `HegicStrategy`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the `Hegic Herge` protocol, the `HegicStrategy` contract implements the standard option trading strategy, while some other contracts inheriting from it implement the specific option trading strategies. In particular, the internal `_create()` routine called inside the `create()` routine is used to create a new option for the user. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `HegicStrategy` contract. Inside the `_create()` routine, the `calculateNegativepnlAndPositivepnl()` routine is called (line 128) to calculate the `positive PNL` and `negative PNL` for the new option. The first returned value of the `calculateNegativepnlAndPositivepnl()` routine is `negative PNL` and the second returned value is `positive PNL`. However, inside the `_create()` routine, we observe its first returned value is used as `positive PNL` and its second returned value is used as `negative PNL`, which is the opposite of its implementation. Given this, we suggest to improve the implementation as below: `(negativePNL, positivePNL)= calculateNegativepnlAndPositivepnl(amount, period, additional)` (line 128).

```
120     function _create(
121         uint256 id,
122         address, /*holder*/
123         uint256 amount,
124         uint256 period,
125         bytes[] calldata additional
126     ) internal virtual returns (uint32 expiration, uint256 positivePNL, uint256
            negativePNL)
127     {
128         (positivePNL, negativePNL) = calculateNegativepnlAndPositivepnl(
129             amount,
130             period,
131             additional
132         );
133         ...
134     }
135
136     function calculateNegativepnlAndPositivepnl(
137         uint256 amount,
138         uint256 period,
139         bytes[] calldata /*additional*/
140     ) public view virtual override returns (uint128 negativepnl, uint128 positivepnl)
141     {
142         negativepnl = _calculateCollateral(amount, period);
143         positivepnl = _calculateStrategyPremium(amount, period);
144     }
```

Listing 3.3: `HegicStrategy::_create()`

Note another routine, i.e., `HegicInverseStrategy::_create()`, shares the same issue.

Recommendation Correct the implementation of the `_create()` routine as above-mentioned.

Status The issue has been addressed by the following commits: `7e9cea0` and `a46c922`.

3.4 Immutable States If Only Set at Constructor()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `Hegic Herge` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
contract OperationalTreasury is
    IOperationalTreasury,
    AccessControl,
    ReentrancyGuard
{
    ...
    ICoverPool public override coverPool;
    ...
    uint256 public maxLockupPeriod;
}
```

Listing 3.4: `OperationalTreasury`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been addressed by the following commit: `7e9cea0`.

3.5 Suggested Event Generation for Key Operations

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several key operations that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setPricer(IPremiumCalculator value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        pricer = value;
    }

    function setK(uint256 value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        k = value;
    }
```

Listing 3.5: HegicStrategy

With that, we suggest to emit meaningful events for these key operations. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: 7e9cea0.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the Hegic Herge protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
    function isPayoffAvailable(uint256 optionID, address caller) external view override
        returns (bool) {
        return
            (hasRole(EXERCISER_ROLE, caller) &&
                _calculateStrategyPayOff(optionID) > 0)
            block.timestamp > positionExpiration[optionID];
    }
```

Listing 3.6: HegicInverseStrategy::isPayoffAvailable()

```
    function setPricer(IPremiumCalculator value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        pricer = value;
    }

    function setLimit(uint256 value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        lockedLimit = value;
        emit SetLimit(value);
    }

    function setK(uint256 value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        k = value;
    }
```

Listing 3.7: HegicStrategy

```
    function setNextEpochChangingPrice(uint256 value) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        nextEpochChangingPrice = value;
    }
```

Listing 3.8: CoverPool::setNextEpochChangingPrice()

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The `multi-sig` mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team.

4 | Conclusion

In this audit, we have analyzed the design and implementation of `Hegic Herge` protocol, which is an on-chain peer-to-pool options trading protocol built on Ethereum. With the protocol, DeFi and crypto users can trade 24/7, cash-settled, various on-chain ETH and WBTC options trading strategies with no KYC or registration required for trading. It provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current DeFi ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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