SMART CONTRACT AUDIT REPORT

for

Omni Protocol

Prepared By: Xiaomi Huang

PeckShield

October 22, 2023

Document Properties

Client	Omni
Title	Smart Contract Audit Report
Target	Omni
Version	1.0
Author	Xuxian Jiang
Auditors	Jonathan Zhao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 22, 2023	Xuxian Jiang	Final Release
1.0-rc	October 20, 2023	Xuxian Jiang	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 `Omni` 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 Omni

`Omni` is a composable, dynamic, and capital efficient money market primitive. The protocol supports a wide array of collateral and allows to borrow assets with zero fragmentation and maximal capital efficiency. The protocol introduces a novel concept of `risk tranches` for asset pools that allows lenders to opt-in and opt-out of lending to certain collateral assets, so lenders earn the maximum yield for their risk profile and borrowers have access to maximum liquidity. In addition, the protocol introduces collision-free sub-accounts for asset management, high efficiency borrowing modes, a joint risk and utilization interest model, timed collateral, proportional loss socialization, and dynamic liquidations. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Omni

Item	Description
Target	Omni
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the `Omni` protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/beta-finance/Omni.git (c440645)

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

- https://github.com/beta-finance/Omni.git (5152a89)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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.

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

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 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) [8], 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 `Omni` 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	0	
Medium	2	■■
Low	3	■■■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Omni Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inaccurate Interest Accrual Logic in OmniToken	Business Logic	Resolved
PVE-002	Low	Improved Liquidation Logic in Omni-Token	Coding Practices	Resolved
PVE-003	Low	Improved enterMode() Logic in Om-niPool	Business Logic	Resolved
PVE-004	Low	Market Deduplication in Om-niPool::enterMarkets()	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Inaccurate Interest Accrual Logic in OmniToken

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OmniToken`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Omni` protocol, the `OmniToken` contract manages deposits, withdrawals, borrowings, and repayments with necessary interest accrual and distribution. While examining the interest accrual logic, we observe its current implementation needs to be revisited.

To elaborate, we show below the implementation of the related `accrue()` routine. As the name indicates, it accrues interest for all tranches by calculating and distributing the interest among the depositors and updating tranche balances. Note that the accrued interest for each tranche will be eventually reflected in the tranche balance. However, it comes to our attention that the current implementation may accidentally overwrite the tranche balance by removing the interest accrued from previous iterations (line 130). To fix, there is a need to replace the statement at line 130 as `tranches[ti].totalDepositAmount += interestAmountProportion`.

```
91      function accrue() public {
92          uint256 timePassed = block.timestamp - lastAccrualTime;
93          if (timePassed == 0) {
94              return;
95          }
96          uint8 trancheIndex = trancheCount;
97          uint256 totalBorrow = 0;
98          uint256 totalDeposit = 0;
99          uint256[] memory trancheDepositAmounts_ = new uint256[](trancheCount);
100         while (trancheIndex != 0) {
101             unchecked {
102                 --trancheIndex;
```

```
            }
            OmniTokenTranche storage tranche = tranches[trancheIndex];
            uint256 trancheDepositAmount_ = tranche.totalDepositAmount;
            uint256 trancheBorrowAmount_ = tranche.totalBorrowAmount;
            totalBorrow += trancheBorrowAmount_;
            totalDeposit += trancheDepositAmount_;
            trancheDepositAmounts_[trancheIndex] = trancheDepositAmount_;

            if (trancheBorrowAmount_ == 0) {
                continue;
            }
            uint256 interestAmount;
            {
                uint256 interestRate = IIRM(irm).getInterestRate(address(this), trancheIndex, totalDeposit, totalBorrow);
                interestAmount = (trancheBorrowAmount_ * interestRate * timePassed) / 365 days / IRM_SCALE;
            }

            // Handle reserve payments
            uint256 reserveInterestAmount = interestAmount * RESERVE_FEE / FEE_SCALE;

            interestAmount -= reserveInterestAmount;
            // Handle deposit interest
            {
                uint256 depositInterestAmount = 0;
                uint256 interestAmountProportion;
                for (uint8 ti = trancheCount - 1; ti > trancheIndex; ti--) {
                    interestAmountProportion = interestAmount * trancheDepositAmounts_[ti] / totalDeposit;
                    tranches[ti].totalDepositAmount = trancheDepositAmounts_[ti] + interestAmountProportion;
                    depositInterestAmount += interestAmountProportion;
                }
                // For the last tranche, we need to add the reserve interest amount to the deposit interest amount
                interestAmountProportion = interestAmount * trancheDepositAmount_ / totalDeposit;
                depositInterestAmount += interestAmountProportion;
                tranche.totalDepositAmount = trancheDepositAmount_ + interestAmountProportion + reserveInterestAmount;
                tranche.totalBorrowAmount = trancheBorrowAmount_ + depositInterestAmount + reserveInterestAmount;
            }

            // Pay reserve fee
            uint256 reserveShare;
            uint256 totalDepositShare_ = tranche.totalDepositShare;
            if (trancheDepositAmount_ == 0) {
                reserveShare = reserveInterestAmount;
            } else {
                reserveShare = (reserveInterestAmount * totalDepositShare_) /
```

```
                    trancheDepositAmount_; // Cannot divide by 0
            }
            trancheAccountDepositShares[trancheIndex][reserveReceiver] += reserveShare;
            tranche.totalDepositShare = totalDepositShare_ + reserveShare;
        }
        lastAccrualTime = block.timestamp;
        emit Accrue();
    }
```

Listing 3.1: `OmniToken::accrue()`

Recommendation Properly update the tranche balance with the accrued interest.

Status The issue has been addressed in the following PR: `1`.

3.2 Improved Liquidation Logic in OmniToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OmniToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

Description

In the `Omni` protocol, the built-in liquidation logic will seize certain collateral from the underwater account for the debt repayment. Our analysis shows the seize amount calculation logic needs to be improved.

To elaborate, we show below the related code snippet of the `seize()` routine. It allows a liquidator to seize funds from a user's account. By design, it may greedily seize as much collateral as possible and does not revert if no more collateral is left to seize. We notice the attempted seize amount of collateral is calculated as `amount = (share * totalAmount)/ totalShare` (line 296), which does not consider the possibility of `totalShare=0`. If there is zero `totalShare`, this `seize()` routine will simply revert the execution. With that, we suggest to add a `continue` statement if that is the case.

```
282    function seize(bytes32 _account, bytes32 _to, uint256 _amount)
283        external
284        override
285        nonReentrant
286        returns (uint256[] memory)
287    {
288        require(msg.sender == omniPool, "OmniToken::seize: Bad caller");
289        accrue();
290        uint256 amount_ = _amount;
291        uint256[] memory seizedShares = new uint256[](trancheCount);
```

```
        for (uint8 ti = 0; ti < trancheCount; ++ti) {
            uint256 totalShare = tranches[ti].totalDepositShare;
            uint256 totalAmount = tranches[ti].totalDepositAmount;
            uint256 share = trancheAccountDepositShares[ti][_account];
            uint256 amount = (share * totalAmount) / totalShare;
            if (amount_ > amount) {
                amount_ -= amount;
                trancheAccountDepositShares[ti][_account] = 0;
                trancheAccountDepositShares[ti][_to] += share;
                seizedShares[ti] = share;
            } else {
                uint256 transferShare = (share * amount_) / amount;
                trancheAccountDepositShares[ti][_account] = share - transferShare;
                trancheAccountDepositShares[ti][_to] += transferShare;
                seizedShares[ti] = transferShare;
                break;
            }
        }
        emit Seize(_account, _to, _amount, seizedShares);
        return seizedShares;
    }
```

Listing 3.2: `OmniToken::seize()`

Recommendation Improve the above `seize()` routine to handle the corner case of `totalShare=0`.

Status The issue has been addressed in the following PR: `1`.

3.3 Improved enterMode() Logic in OmniPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OmniPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Omni` protocol is able to make better risk assumptions when it knows specific assets are being used for lending and borrowing. This feature is similar to the high efficiency modes in `AaveV3`, where if a borrower is only using stablecoins then they get more favorable loan terms. While examining the mode-entering logic, we notice the implementation needs to be improved.

In the following, we show the related code snippet from the `enterMode()` routine. In order to enter the intended high efficiency mode, we need to ensure the given sub-account must not already be in a mode and the mode must not have expired. In the meantime, the given sub-account should also not enter any isolated market. Otherwise, the exclusion among mode, isolation market, as well as other generic markets is broken.

```
    function enterMode(uint96 _subId, uint8 _modeId) external {
        bytes32 accountId = msg.sender.toAccount(_subId);
        require(_modeId > 0 && _modeId <= modeCount, "OmniPool::enterMode: Invalid mode ID.");
        AccountInfo memory account = accountInfos[accountId];
        require(account.modeId == 0, "OmniPool::enterMode: Already in a mode.");
        require(accountMarkets[accountId].length == 0, "OmniPool::enterMode: Non-zero market count.");
        require(modeConfigurations[_modeId].expirationTimestamp > block.timestamp, "OmniPool::enterMode: Mode expired.");
        account.modeId = _modeId;
        accountInfos[accountId] = account;
        emit EnteredMode(accountId, _modeId);
    }
```

Listing 3.3: `OmniPool::enterMode()`

Recommendation Revise the above `enterMode()` logic to ensure the given account does not have entered any isolated market.

Status The issue has been addressed in the following PR: `1`.

3.4 Market Deduplication in OmniPool::enterMarkets()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OmniPool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

As mentioned earlier, `Omni` allows the user to enter various markets being supported in the protocol. While examining the market-entering logic, we notice there is a market deduplication requirement that is not being properly enforced.

To elaborate, we show below the implementation of the related `enterMarkets()` routine. In order to enter multiple unique markets, there is a need to ensure none of them is an isolated collateral market. Also, there is an implicit requirement that the given markets should be unique. However, the uniqueness enforcement (line 131) is flawed and should be corrected as `require(!_contains(newMarkets, market))`.

```
    function enterMarkets(uint96 _subId, address[] calldata _markets) external {
        bytes32 accountId = msg.sender.toAccount(_subId);
        require(accountInfos[accountId].modeId == 0, "OmniPool::enterMarkets: Already in a mode.");
        address[] memory existingMarkets = accountMarkets[accountId];
        address[] memory newMarkets = new address[](existingMarkets.length + _markets.length);
        for (uint256 i = 0; i < existingMarkets.length; ++i) {
            newMarkets[i] = existingMarkets[i];
        }
        for (uint256 i = 0; i < _markets.length; ++i) {
            address market = _markets[i];
            MarketConfiguration memory marketConfig = marketConfigurations[market];
            require(
                marketConfig.expirationTimestamp > block.timestamp && !marketConfig.isIsolatedCollateral,
                "OmniPool::enterMarkets: Market invalid."
            );
            require(!_contains(existingMarkets, market), "OmniPool::enterMarkets: Already in the market.");
            require(
                IOmniToken(market).getBorrowCap(0) > 0,
                "OmniPool::enterMarkets: Market has no borrow cap for 0 tranche."
            );
            newMarkets[i + existingMarkets.length] = market;
        }
        accountMarkets[accountId] = newMarkets;
        emit EnteredMarkets(accountId, _markets);
    }
```

Listing 3.4: `OmniPool::enterMarkets()`

Recommendation Ensure the entered markets are unique and do not contain any duplicate market.

Status The issue has been addressed in the following PR: `1`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

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

```
    function setTrancheCount(address _market, uint8 _trancheCount) external onlyRole(
        MARKET_CONFIGURATOR_ROLE) {
        IOmniToken(_market).setTrancheCount(_trancheCount);
    }

    /**
     * @notice Sets the borrow cap for each tranche of a specific market.
     * @dev This function can only be called by an account with the
         MARKET_CONFIGURATOR_ROLE.
     * It invokes the setTrancheBorrowCaps function of the IOmniToken contract
         associated with the specified market.
     * @param _market The address of the market for which to set the borrow caps.
     * @param _borrowCaps An array of borrow cap values, one for each tranche of the
         market.
     */
    function setBorrowCap(address _market, uint256[] calldata _borrowCaps)
        external
        onlyRole(MARKET_CONFIGURATOR_ROLE)
    {
        IOmniToken(_market).setTrancheBorrowCaps(_borrowCaps);
    }

    /**
     * @notice Sets the supply cap for a market that doesn't allow borrowing.
     * @dev This function can only be called by an account with the
         MARKET_CONFIGURATOR_ROLE.
     * It invokes the setSupplyCap function of the IOmniTokenNoBorrow contract
         associated with the specified market.
     * @param _market The address of the market for which to set the no-borrow supply
         cap.
     * @param _noBorrowSupplyCap The value of the no-borrow supply cap to set.
     */
    function setNoBorrowSupplyCap(address _market, uint256 _noBorrowSupplyCap)
        external
```

```
        onlyRole(MARKET_CONFIGURATOR_ROLE)
    {
        IOmniTokenNoBorrow(_market).setSupplyCap(_noBorrowSupplyCap);
    }

    /**
     * @notice Sets the reserve receiver's address. This function can only be called by an account with the DEFAULT_ADMIN_ROLE.
     * @dev The reserve receiver's address is converted to a bytes32 account identifier using the toAccount function with a subId of 0.
     * @param _reserveReceiver The address of the reserve receiver to be set.
     */
    function setReserveReceiver(address _reserveReceiver) external onlyRole(DEFAULT_ADMIN_ROLE) {
        reserveReceiver = _reserveReceiver.toAccount(0);
    }

    /**
     * @notice Pauses the protocol, halting certain functionalities, i.e. withdraw, borrow, repay, liquidate.
     * @dev This function triggers the `_pause()` internal function and sets `pauseTranche` to 0.
     * It's an external function that can only be called by an account with the `DEFAULT_ADMIN_ROLE`.
     * The function can only be executed when the contract is not already paused,
     * which is checked by the `whenNotPaused` modifier.
     */
    function pause() external whenNotPaused onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
        pauseTranche = 0;
    }
```

Listing 3.5: Example Privileged Operations in `OmniPool`

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 mitigated with the use of `4 of 7 multisig` to possess the admin role.

4 | Conclusion

In this audit, we have analyzed the design and implementation of `Omni`, which is a composable, dynamic, and capital efficient money market primitive. The protocol supports a wide array of collateral and allows to borrow assets with zero fragmentation and maximal capital efficiency. The protocol introduces a novel concept of `risk tranches` for asset pools that allows lenders to opt-in and opt-out of lending to certain collateral assets, so lenders earn the maximum yield for their risk profile and borrowers have access to maximum liquidity. In addition, the protocol introduces collision-free sub-accounts for asset management, high efficiency borrowing modes, a joint risk and utilization interest model, timed collateral, proportional loss socialization, and dynamic liquidations. 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-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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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