SMART CONTRACT AUDIT REPORT

for

TranchessV2 Protocol

Prepared By: Xiaomi Huang

PeckShield

May 20, 2022

Document Properties

Client	Tranchess Protocol
Title	Smart Contract Audit Report
Target	TranchessV2
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 20, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 10, 2022	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 `TranchessV2` design document and related smart contract source code, 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 Tranchess

Elevating from the current tranche-fund model, `TranchessV2` aims to improve the liquidity and accessibility of the current `BISHOP/ROOK/QUEEN` token, through a newly introduced stable swap `AMM` pool. In contrast to `V1`, where users have to cross the bid-ask spread to trade in an orderbook system, or wait up to 24 hours for the creation of `QUEEN` token, the `V2` `AMM` pool allows users to freely convert from `BUSD/BTC/ETH/BNB` into `BISHOP/ROOK/QUEEN`. This `AMM` swap process is instantaneous and more cost efficient. It not only allows `ROOK` holders to timely enter and exit their leverage trading positions, but also enables `BISHOP` holders to earn extra yield through liquidity provision in the `BISHOP-BUSD` pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TranchessV2

Item	Description
Name	Tranchess Protocol
Website	https://tranchess.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 20, 2022

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

- https://github.com/tranchess/contract-core.git (6876889)

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

- https://github.com/tranchess/contract-core.git (bcc1ad0)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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

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 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) [9], 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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `TranchessV2` 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	1	■
Low	2	■ ■
Informational	3	■ ■ ■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key TranchessV2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in SwapReward::updateReward()	Business Logic	Resolved
PVE-002	Low	Revisited Logic in InterestRateBallot::syncWithVotingEscrow()	Business Logic	Resolved
PVE-003	Informational	Revisited Reentrancy Protection in VestingEscrow	Time And State	Resolved
PVE-004	Informational	Redundant State/Code Removal	Coding Practices	Resolved
PVE-005	Informational	Consistent Fee Calculation in StableSwap	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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

3 | Detailed Results

3.1 Improved Logic in SwapReward::updateReward()

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

Description

The `TranchessV2` protocol has a helper `SwapReward` contract that assists the reward dissemination to the associated `liquidityGauge`. While examining a number of parameters that are updated in `SwapReward`, we notice the update logic can be improved to accommodate different configurations.

Specifically, we show below the related `updateReward()` function from the `SwapReward` contract. It has a rather straightforward logic in assigning the input arguments to internal storage states. However, it comes to our attention that the given `start`, `interval`, `amount`, and `ratePerSecond` may have internal dependency. As a result, we need to properly compute `ratePerSecond` within the resulting interval between `startTimestamp` and `endTimestamp` where `startTimestamp = start.max(block.timestamp)` and `endTimestamp = start.add(interval)`. In other words, we can compute `ratePerSecond = amount.div( endTimestamp.sub(startTimestamp))`.

```
    function updateReward(
        address caller,
        uint256 amount,
        uint256 start,
        uint256 interval
    ) external onlyOwner {
        require(
            endTimestamp < block.timestamp && endTimestamp == lastTimestamp,
            "Last reward not yet expired"
        );
        require(caller != address(0));
        ratePerSecond = amount.div(interval);
```

```
        startTimestamp = start.max(block.timestamp);
        endTimestamp = start.add(interval);
        lastTimestamp = startTimestamp;
        IERC20(rewardToken).safeTransferFrom(
            msg.sender,
            address(this),
            ratePerSecond.mul(interval)
        );
    }
```

Listing 3.1: `SwapReward::updateReward()`

Recommendation Properly compute the `ratePerSecond` from the given input arguments in the above `updateReward()` function.

Status The issue has been fixed by the following commit: `78c5fbdc`.

3.2 Revisited Logic in InterestRateBallot::syncWithVotingEscrow()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `InterestRateBallot`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `TranchessV2` protocol has primary markets that are designed to allow users to create, redeem, split, or merge `TRANCHE_Q` tokens. In the meantime, it also has a `InterestRateBallot` contract that allows to adjust the interest rate. This `InterestRateBallot` contract will need to timely synchronize with the `VotingEscrowV2` contract to keep track of the voter amount for related users. While examining the synchronization logic, we notice the current implementation can be improved.

To elaborate, we show below the related `syncWithVotingEscrow()` function, which updates the vote amount for the given account. It comes to our attention that the current logic simply returns when the user does not have any locked balance (with 0 amount or expired position). Our analysis shows that there is a need to continue the execution to reset the voter amount. Note that the same issue is also applicable to other contracts, including `FeeDistributor`.

```
    function syncWithVotingEscrow(address account) external override {
        Voter memory voter = voters[account];
        if (voter.amount == 0) {
            return; // The account did not voted before
```

```
        }

        IVotingEscrow.LockedBalance memory lockedBalance = votingEscrow.getLockedBalance(account);
        if (lockedBalance.amount == 0  lockedBalance.unlockTime <= block.timestamp) {
            return;
        }

        // update scheduled unlock
        scheduledUnlock[voter.unlockTime] = scheduledUnlock[voter.unlockTime].sub(voter.amount);
        scheduledUnlock[lockedBalance.unlockTime] = scheduledUnlock[lockedBalance.unlockTime].add(
            lockedBalance.amount
        );

        scheduledWeightedUnlock[voter.unlockTime] = scheduledWeightedUnlock[voter.unlockTime].sub(
            voter.amount * voter.weight
        );
        scheduledWeightedUnlock[lockedBalance.unlockTime] = scheduledWeightedUnlock[
            lockedBalance.unlockTime
        ]
            .add(lockedBalance.amount * voter.weight);

        emit Voted(
            account,
            voter.amount,
            voter.unlockTime,
            voter.weight,
            lockedBalance.amount,
            lockedBalance.unlockTime,
            voter.weight
        );

        // update voter amount per account
        voters[account].amount = lockedBalance.amount;
        voters[account].unlockTime = lockedBalance.unlockTime;
    }
```

Listing 3.2: `InterestRateBallot::syncWithVotingEscrow()`

Recommendation Properly revise the above logic to update the voter amount, including the cases where the user may have a 0 balance or expired lock position.

Status The issue has been fixed by the following commit: `3d0b6c4`.

3.3 Revisited Reentrancy Protection in VestingEscrow

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: VestingEscrow
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice the current protocol has taken into consideration the re-entrancy protection in current contracts. For example, the nonReentrant modifier is widely used, even in certain cases where it is safe to not use the re-entrancy protection. For example, we show below the VestingEscrow::claim() function, which allows to claim the tokens that were vested before. Our analysis shows that the associated nonReentrant modifier is not necessary as it follows strictly the above checks-effects-interactions principle with the token-transfer at the end of this function.

```
    /// @notice Claim tokens which have vested
    function claim() external nonReentrant {
        uint256 timestamp = disabledAt;
        if (timestamp == 0) {
            timestamp = block.timestamp;
        }
        uint256 claimable = _totalVestedOf(timestamp).sub(totalClaimed);
        totalClaimed = totalClaimed.add(claimable);
        IERC20(token).safeTransfer(recipient, claimable);

        emit Claim(claimable);
    }
```

Listing 3.3: VestingEscrow::claim()

Recommendation Remove the unnecessary nonReentrant modifier in the above claim() function.

Status The issue has been fixed by the following commit: 9a211be.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: FundV3
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

The TranchessV2 protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20 , Ownable, and ReentrancyGuard, to facilitate its code implementation and organization. For example, the FundV3 smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the FundV3 contract, there are a number of events that are defined, but not used. Examples include the ObsoletePrimaryMarketAdded and NewPrimaryMarketAdded structures.

```
    event ProfitReported(uint256 profit, uint256 performanceFee);
    event LossReported(uint256 loss);
    event ObsoletePrimaryMarketAdded(address obsoletePrimaryMarket);
    event NewPrimaryMarketAdded(address newPrimaryMarket);
    event DailyProtocolFeeRateUpdated(uint256 newDailyProtocolFeeRate);
    event TwapOracleUpdated(address newTwapOracle);
    event AprOracleUpdated(address newAprOracle);
    event BallotUpdated(address newBallot);
    event FeeCollectorUpdated(address newFeeCollector);
    event ActivityDelayTimeUpdated(uint256 delayTime);
    event SplitRatioUpdated(uint256 newSplitRatio);
    event FeeDebtPaid(uint256 amount);
```

Listing 3.4: Various Events Defined in FundV3

Recommendation Consider the removal of the non-used events with a simplified, consistent implementation.

Status The issue has been fixed by the following commit: bcc1ad0.

3.5 Consistent Fee Calculation in StableSwap

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StableSwap`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `TranchessV2` protocol has an innovative `StableSwap AMM` contract, which allows users to freely convert from `BUSD/BTC/ETH/BNB` into `BISHOP/ROOK/QUEEN` without the need of crossing the bid-ask spread to trade in an orderbook system, or waiting up to 24 hours for the creation of `QUEEN` token. The new `AMM` has the built-in support of collecting swap fee and protocol fee. Our analysis shows the fee collection may be improved.

To elaborate, we show below the related `getQuoteOut()/getQuoteIn()` functions. As their names indicate, these two functions are proposed to calculate the quote amount with the given `baseIn` or the expected `baseOut`. It comes to our attention that the first function computes the fee on the output token, while the second function computes the fee on the input token. The other two functions `getBaseOut()` and `getBaseIn()` share the same issue.

```
    function getQuoteOut(uint256 baseIn) external view override returns (uint256 quoteOut) {
        (uint256 oldBase, uint256 oldQuote, , , , , ) =
            _getRebalanceResult(fund.getRebalanceSize());
        uint256 newBase = oldBase.add(baseIn);
        uint256 ampl = getAmpl();
        uint256 oraclePrice = getOraclePrice();
        uint256 d = _getD(oldBase, oldQuote, ampl, oraclePrice);
        uint256 newQuote = _getQuote(ampl, newBase, oraclePrice, d);
        quoteOut = oldQuote.sub(newQuote).sub(1); // -1 just in case there were some rounding errors
        uint256 fee = quoteOut.multiplyDecimal(feeRate);
        quoteOut = quoteOut.sub(fee);
    }

    function getQuoteIn(uint256 baseOut) external view override returns (uint256 quoteIn) {
        (uint256 oldBase, uint256 oldQuote, , , , , ) =
            _getRebalanceResult(fund.getRebalanceSize());
        uint256 newBase = oldBase.sub(baseOut);
        uint256 ampl = getAmpl();
        uint256 oraclePrice = getOraclePrice();
        uint256 d = _getD(oldBase, oldQuote, ampl, oraclePrice);
        uint256 newQuote = _getQuote(ampl, newBase, oraclePrice, d);
```

```
        quoteIn = newQuote.sub(oldQuote).add(1); // 1 just in case there were some
            rounding errors
        uint256 fee = quoteIn.mul(feeRate).div(uint256(1e18).sub(feeRate));
        quoteIn = quoteIn.add(fee);
    }
```

Listing 3.5: `StableSwap::getQuoteOut()/getQuoteIn()`

Recommendation Properly compute the fee consistently in the above set of functions.

Status The issue has been resolved as the team confirms it is one of the design choices, which makes it unique comparing to `UNI` or `Curve`. In particular, since the base asset is either `BISHOP` or `QUEEN` and both subject to rebalance, the team feels it better to collect all swap fees only in quote asset, making storing and distributing of the fees much easier.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `TranchessV2` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function _updateFundCap(uint256 newCap) private {
        fundCap = newCap;
        emit FundCapUpdated(newCap);
    }

    function updateFundCap(uint256 newCap) external onlyOwner {
        _updateFundCap(newCap);
    }

    function _updateRedemptionFeeRate(uint256 newRedemptionFeeRate) private {
        require(newRedemptionFeeRate <= MAX_REDEMPTION_FEE_RATE, "Exceed max redemption
            fee rate");
        redemptionFeeRate = newRedemptionFeeRate;
        emit RedemptionFeeRateUpdated(newRedemptionFeeRate);
```

```
    }

    function updateRedemptionFeeRate(uint256 newRedemptionFeeRate) external onlyOwner {
        _updateRedemptionFeeRate(newRedemptionFeeRate);
    }

    function _updateMergeFeeRate(uint256 newMergeFeeRate) private {
        require(newMergeFeeRate <= MAX_MERGE_FEE_RATE, "Exceed max merge fee rate");
        mergeFeeRate = newMergeFeeRate;
        emit MergeFeeRateUpdated(newMergeFeeRate);
    }

    function updateMergeFeeRate(uint256 newMergeFeeRate) external onlyOwner {
        _updateMergeFeeRate(newMergeFeeRate);
    }
```

Listing 3.6: Example Privileged Operations in the `PrimaryMarketV3` Contract

In addition, we notice the `owner` account that is able to adjust various protocol-wide risk parameters. Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that if current contracts need to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 This issue has been confirmed and partially mitigated. Especially, for all admin-level operations, the current mitigation is to adopt the standard `TimelockController` with multi-sig `TranchessV2` account as the proposer, and a minimum delay of 1 days. The `TimelockController` address on BSC chain is `0x4BB3AeB5Ba75bC6A44177907B54911b19d1cF8f7`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `TranchessV2` protocol, which greatly improves the liquidity and accessibility of the current `BISHOP/ROOK/QUEEN` token, through a newly introduced stable swap `AMM` pool. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.